Hornig · Weltner · Trapp

COMMODORE 128

Tips & Tricks

Eine wahre Fundgrube für den 128er Anwender

EIN DATA BECKER BUCH

DAS STEHT DRIN:

128 Tips & Tricks ist eine riesige Fundgrube für jeden 128er Besitzer, der mehr mit seinem Rechner machen will. Dieses Buch enthält nicht nur viele Beispielprogramme, sondern erläutert auch leicht verständlich den Aufbau des Rechners und seine Programmierung.

Aus dem Inhalt:

- Grafik auf dem Commodore 128
- Arbeiten mit mehreren Bildschirmen
- Eigener Zeichensatz
- Sprite-Handling
- Grafik mit den eingebauten Befehlen
- Simulation mehrerer Windows
- Listing-Konverter
- Modifiziertes Input
- Software-Schutz auf dem Commodore 128
- Zeilen einfügen
- Rund um die Tastatur
- Befehlserweiterung - selbst gemacht
- Banking
- Weitere Möglichkeiten der MMU
- Autostart
- Der Speicher
- Wechseln des Betriebssystems
- Der 64er-Modus auf dem C-128
- Die 10er Tastatur am C-64
  und vieles mehr

HEIMCOMPUTER-ZU
BEHOER DIVERSE

A4349999+0047.00

061 47.00

UND GESCHRIEBEN HAT DIESES BUCH:

Das Autorenteam mit Tobias Weltner, Ralf Hornig und Jens Trapp (64 Tips & Tricks Band II) arbeitet mit dem 128er, seit es diesen Rechner gibt. Alle sind begeisterte Programmierer, die Ihre gesamte Erfahrung in dieses Buch gesteckt haben.

ISBN 3-89011-097-5

Hornig · Weltner · Trapp

COMMODORE 128

Tips & Tricks

Eine wahre Fundgrube für den 128er Anwender

EIN DATA BECKER BUCH

ISBN 3-89011-097-5

Copyright © 1985 DATA BECKER GmbH
Merowingerstraße 30
4000 Düsseldorf

Alle Rechte vorbehalten. Kein Teil dieses Buches darf in irgendeiner Form (Druck, Fotokopie oder einem anderen Verfahren) ohne schriftliche Genehmigung der DATA BECKER GmbH reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Schaltungen, Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle Schaltungen, technischen Angaben und Programme in diesem Buch wurden von dem Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.

0. Vorwort 1

1. Grafik auf dem Commodore 128 3
1.1. Umschalten: 40 / 80 Zeichen 5
1.2. Der 40-Zeichen-Bildschirm 5
1.3. Der 40-Zeichen-Charactergenerator 6
1.3.1. Verändern des Zeichensatzes 11
1.3.2. Zeichen-Editor 12
1.4. Verschieben des Bildschirmspeichers 13
1.4.1. Arbeiten mit mehreren Bildschirmen 13
1.5. Der 80-Zeichen-Bildschirm 15
1.6. Bildschirm- und Farb-RAM 16
1.7. Registerbeschreibung des 80-Zeichen-Controllers 16
1.8. Der Display-Controller 17
1.9. Zuverlässiger Video-RAM-Zugriff 20
1.10. POKE-Simulation 22
1.11. Der Charactergenerator 24
1.12. Auslesen des Zeichengenerators 24
1.13. Big Script mit Strings 25
1.14. Transparente drucken für den Heimgebrauch 26
1.15. Definition eines eigenen Zeichensatzes 26
1.16. Mit dem Zeichen-Editor: Definition leicht gemacht 29
1.17. Arbeiten mit mehreren Bildschirmen 33
1.18. Sinnvolle Manipulationen des VCD 8563 37
1.19. Manipulation des Bildschirmformats 39
1.20. So geht's effektiver für Grünmonitor-Besitzer 41
1.21. Der erste Schritt zur neuen Matrix 43
1.22. Echte Manipulation der Punktematrix 44
1.23. Doppelt hohe Zeichen 46
1.24. Interne Verschiebungen im Video-RAM 49
1.25. Farbe für den 80-Zeichen-Schirm 52
1.26. Ein zusätzlicher Zeichengenerator 54
1.27. Systemroutinen für den Hausgebrauch 58
1.28. Hochauflösende Grafik 62
1.29. Zeichengenerator einmal anders 65

2. Grafik mit den eingebauten Befehlen 67
2.1. Der CIRCLE-Befehle 68
2.2. Tortengrafik 69
2.3. Die Balkengrafik 70
2.4. Funktionsplotter 71
2.5. Windows 72
2.5.1. Wie man ein Fenster verläßt 72
2.5.2. Abfragen der Window-Koordinaten 73
2.5.3. Alternatives Fenster setzen 74
2.5.4. Vertikale Laufschrift 76
2.5.5. Das Fenster als Eingabebegrenzung 76
2.5.6. PRINT AT mit Windows 77
2.5.7. Löschen eines Teil-Bildschirms 78
2.5.8. Window-Inhalt sichern 79
2.5.9. Simulation mehrerer Fenster 82
2.6. Sprite-Handling 83
2.6.1. Design im Listing 84
2.6.2. Komfortable Sprite-Editierung 85

3. Nützliche Programme 87
3.1. Fehlerbehandlung 88
3.2. Listing-Konverter 94
3.3. OLD 99
3.4. Musik nebenbei 101
3.5. Echtzeituhr auf dem Commodore 128 104
3.6. Analoguhr 106
3.7. LLIST 107
3.8. Textverarbeitung Marke Eigenbau 108
3.9. Modifiziertes Input 110
3.10. Signalton 111

4. Software-Schutz auf dem Commodore 128 115
4.1. Schutz durch Doppelpunkte 116
4.2. Zeilennummern-Roulette 118
4.3. Manipulation des Zeilen-Links 120
4.4. Künstliche Steuerzeichen: ein kleiner Kobold 121
4.5. Schutz durch Pokes 123
4.5.1. Verhindern des LIST-Befehls 123
4.5.2. Ausschalten von RUN-STOP / RESTORE 124

4.5.3. Verhindern von SAVE 125
4.6. Kopierschutz für Disk 126
4.7. Directory-Klauer 126

5. Das System der 1000 Möglichkeiten: Zeilen einfügen 129
5.1. Anpaßbare Rechenroutine 132
5.2. DATA-Generator 133

6. Der Datenrekorder 137
6.1. Software-Steuerung der Datasette 138
6.2. Abfrage der Bandtasten 139
6.3. Ungewöhnliches mit dem Datenrichtungsregister 140
6.4. Ein etwas anderer Kopierschutz 141
6.5. "HiFi"-Klänge - die Datasette als begnadete "Musikbox" 142
6.6. Saven auf Datasette - einmal anders! 144

7. Rund um die Tastatur 147
7.1. Die Tastaturbelegung 148
7.2. Ändern der Tastaturbelegung 151
7.3. Hextastatur für den C-128 154
7.4. Doppelbelegungen 155
7.5. Die Hilftasten 159
7.5.1. Nutzung der Hilfstasten 159
7.6. Vier zusätzliche Funktionstasten 160
7.7. Tastatur-Piep 162
7.8. Die STOP-Taste 163
7.9. Belegung der HELP-Taste 165

8. Befehlserweiterung - selbst gemacht 167
8.1. Was ist die CHRGET-Routine? 169
8.2. Wie verändert man die CHRGET-Routine? 172
8.3. Das "Verhalten" der neuen Befehle 177
8.4. Mehrere zusätzliche Befehle 178

9. Banking 183
9.1. Theoretische Grundlagen 184
9.2. Banking beim C-128 185
9.3. Umschalten der Banks über die MMU 186
9.4. Weitere Möglichkeiten der MMU 189

10. Autostart 191
10.1. Autostart mit der Floppy 192
10.1.1. Die Routine boot-call 193
10.1.2. Die Verwendung von boot-call 206
10.2. Autostart bei Modulen 212

11. Der Speicher 217
11.1. Nützliche Adressen des C-128 218
11.2. Sprungtabellen 234
11.2.1. Kernal 234
11.2.2. Vektor-Lade-Tabelle 247
11.2.3. Kernal-Aufruf 248
11.3. Freier Speicher 249
11.3.1. Verwendbare Adressen in der Zeropage 249
11.3.2. Verwendbarer Speicher für Maschinenprogramme 250

12. Wechseln des Betriebssystemes 253

13. Der 64er-Modus auf dem C-128 259
13.1. High-Speed für den C-64 260
13.2. Der Zugriff auf den 80-Zeichen-Controller 262
13.3. Die 10er-Tastatur am C-64 262

14. Anhang 265
14.1. Token-Tabelle 266

15. Programmlistings 273

VORWORT

0. Vorwort

****

Dieses Buch haben wir für jeden Commodore 128 Besitzer geschrieben, der mehr aus seinem Gerät herausholen will. Ob Sie sich einen eigenen Zeichensatz erstellen, die doppelte Rechengeschindigkeit auch für 64er Programme nutzen oder die vorhandenen ROM-Routinen verwenden wollen, in jedem Fall ist dieses Buch randvoll mit wichtigen Informationen: Banking und Speicherkonfigurationen, Register-Erläuterungen zum Video-Controller, Windows, Multitasking, Befehlserweiterungen, wichtige Speicherstellen und vor allem viele, viele Beispielprogramme.
Alle größeren und komplizierteren BASIC-Programme sowie BASIC-Loader entsprechender Maschinensprache-Listings befinden sich am Schluß des Buches. Es sind Original-Listings, die nach dem Test sofort ausgedruckt und original ins Buch übernommen wurden, um möglichst viele Fehlerquellen auszuschalten. Im Text sind die Hinweise auf die Programme fortlaufend durchnummeriert, so daß sie sehr leicht aufzufinden sind. Uns bleibt nur, Ihnen viel Spaß beim Programmieren mit diesem Buch und dem Commodore 128 zu wünschen. Doch bevor Sie jetzt in die Tiefen Ihres 128ers hinabtauchen, sollten Sie Murphy's Gesetze *) der Programmierung kennen:

1. Jedes fertige Programm, das läuft, ist veraltet.
2. Jedes andere Programm kostet mehr und dauert länger.
3. Wenn ein Programm nützlich ist, so wird es durch ein anderes ersetzt.
4. Wenn ein Programm nutzlos ist, so wird es dokumentiert.
5. Jedes fertige Programm wird dazu beitragen, das verfügbare Gedächtnis zu füllen.
6. Der Wert eines Programms steht im Verhältnis zu dem Gewicht seiner Produktion.
7. Die Programmverwicklung wächst so lange, bis sie die Fähigkeit des Programmierers übertrifft, der sie weiterführen muß.

Viel Spaß wünschen die Autoren

Rinteln, den 7. August 1985

*) Quelle: A. Bloch, Der Grund warum alles schiefgeht, was schiefgehen kann, Goldmann 1977

1. GRAFIK AUF DEM COMMODORE 128

1. Grafik auf dem Commodore 128

****

Grafik ist sowohl für Programmierer als auch Anwender einer der interessantesten Themenbereiche. An der Grafikfähigkeit, der Möglichkeit also, hochauflösende Grafiken erstellen zu können, wie auch an der Auflösung dieser Grafik wird ein Rechner oftmals gemessen.
Störten bisher die etwas mickrigen 40 Zeichen pro Zeile den Einsatz professioneller Software auf dem Commodore 64, so ist hier alles anders. Wie vom Commodore 64 her bekannt, finden Sie auf der Rückseite des Rechners zunächst einmal zwei Buchsen, bezeichnet mit "RF" und "Video". Diese Buchsen haben die gleiche Aufgabe wie beim C64 und dienen dem Anschluß eines Fernsehgerätes (RF) oder eines herkömmlichen Composite-Monitors (z.B. 1701). Diese Bildschirme können maximal 40 Zeichen pro Zeile darstellen und entsprechen in wesentlichen Punkten der Bildschirmdarstellung des C64.
Aber es ist noch eine weitere Buchse mit der Bezeichnung "RGB" hinzugekommen! An diese läßt sich ein RGB-Monitor anschließen. Zugegeben: RGB-Monitore sind im Vergleich zu den zuvor erwähnten Bildschirmen teurer. Aber auf ihnen lassen sich bis zu 80 Zeichen pro Zeile darstellen. Auch viele weitere Vorteile, angefangen mit wirklich verbesserter Bildqualität bis hin zu wesentlich höherer Auflösung, lassen uns zu dem Schluß kommen, daß sich die Anschaffung eines solchen Monitors wirklich lohnt!
Auf den folgenden Seiten möchten wir Sie mit den Besonderheiten beider Bildschirmdarstellungen bekannt machen. Sie werden eine Reihe erklärter Programme vorfinden, die Ihnen das Gesagte verdeutlichen. Vielleicht werden Sie sich über den Umfang wundern, den dieses Kapitel hat. Aber erstens sind es gerade die Grafik-Fähigkeiten, die einen deutlichen Unterschied zum C64 bilden. Außerdem verfügt der C128 ja über zwei voneinander völlig unabhängige Bildschirme, die folglich der individuellen Erklärung bedürfen. Nicht zuletzt ist das Thema Grafik aber auch ein so vielseitiges und oft angewendetes, daß dieser Umfang durchaus angemessen ist. Sehen Sie selbst!

1.1 Umschalten: 40/80 Zeichen

Bevor wir nun ins Detail gehen, möchten wir Ihnen mehrere Möglichkeiten an die Hand geben, wie Sie vom 40-Zeichen- in den 80-Zeichen-Bildschirm schalten und umgekehrt.
Welcher Bildschirm nach dem Einschalten aktiv ist, hängt von der Taste "40/80 Display" in der obersten Reihe der Tastatur ab. Diese Taste hat auf die Umstellung des Bildschirmes nach dem Einschalten keinen Einfluß mehr. Sie wirkt nur während des Resets. Wollen Sie also von Ihrem Programm aus zwischen dem 40- und 80-Zeichen-Bildschirm hin- und herschalten, so bieten sich Ihnen folgende Möglichkeiten:

```
ESC + X              (schaltet im Direktmode um)
PRINT CHR$(27)+"X"   (bewirkt die Umschaltung aus einem
                      Programm)
SYS 49194            (bewirkt gleichfalls die Umschaltung
                      von BASIC, jedoch auch von
                      Maschinensprache aus)
```

1.2. Der 40-Zeichen-Bildschirm

Der 40-Zeichen-Bildschirm wird vom VIC 8564 versorgt, dieser entspricht im wesentlichen dem vom Commodore 64 her bekannten VIC 6564, ist allerdings um zwei Register bereichert worden (auf diese wird später noch genauer eingegangen). Ist dieser Bildschirm aktiviert (siehe oben!), so wirken alle PRINT- und ähnliche Befehle auf die an diesen VIC angeschlossenen Geräte wieFernseher und Composite Monitor- NICHT jedoch auf den RGB-Monitor, da dieser einen eigenen Controller besitzt.
Alle Commodore-64-Umsteiger wird es freuen: Die Speicherbereiche für das Video-RAM und den Farbspeicher sind dieselben geblieben:

```
Bildschirm-RAM  $0400 - $07FF   (dez. 1024- 2023)
Farb-RAM        $D800 - $DBFF   (dez.55296-56295)
```

Beide Speicher liegen im normalen Adreßraum des Prozessors und können daher leicht von Ihnen durch PEEK oder POKE manipuliert werden:

```
10 FOR A=0 TO 255
20 POKE 1024+A,A
30 NEXT A
40 :
50 FOR A=0 TO 255
60 X=INT(RND(1)*16): REM Zufallszahl 0-15 (Farbe)
70 POKE 55296+A,X
80 NEXT A
```

Grundsätzlich gilt also:

POKE 1024+Spalte+(40*Zeile),0-255
setzt ein Zeichen auf den 40-Zeichen-Bildschirm

POKE 55296+Spalte+(40*Zeile),0-15
versieht die Speicherposition mit entsprechender Farbe

Spalte: 0-39 Zeile: 0-24

Analog zum C-64 gilt auch, daß der Bildschirmspeicher in 1k-Schritten im gesamten Speicher verschoben werden kann, der Farbspeicher jedoch unveränderlich ist.

1.3. Der 40-Zeichen-Charactergenerator

Die äußere Gestalt aller Zeichen auf dem Bildschirm muß sinnigerweise irgendwo gespeichert sein. Dieser Speicherbereich wird "Character-" oder auch "Zeichengenerator" genannt.
Beim Commodore 128 liegt dieser Speicherbereich wie auch beim C64 im Bereich $D000 bis $DFFF im ROM. Diesen Bereich muß sich der Zeichengenerator aber mit dem I/O-Block teilen. Man kann daher nicht ohne weiteres den Zeichengenerator lesen (da er im ROM liegt, kann er ohnehin nicht beschrieben werden!).

So sieht die interne Aufteilung des Zeichengenerators aus:

```
Zeichensatz 1 (Groß-/Blockgrafik-Zeichen)
-------------------------------------------------------
$D000 - $D1FF Großbuchstaben
$D200 - $D3FF Blockgrafik-Zeichen
$D400 - $D5FF Großbuchstaben revers
$D600 - $D7FF Blockgrafik-Zeichen revers
-------------------------------------------------------

Zeichensatz 2 (Klein-/Großbuchstaben)
-------------------------------------------------------
$D800 - $D9FF Kleinbuchstaben
$DA00 - $DBFF Großbuchstaben
$DC00 - $DDFF Kleinbuchstaben revers
$DE00 - $DFFF Großbuchstaben revers
-------------------------------------------------------
```

Das folgende kleine Programm setzt die Speicherkonfiguration so um, daß der Charactergenerator ausgelesen werden kann. Anschließend wird der gelesene Zeichensatz dargestellt:

(Programm 1.3.a)

Das Äußere jedes Zeichens ist in acht Bytes gespeichert. Jedes Byte besteht wiederum aus acht Bits. Insgesamt stehen pro Zeichen also 8*8=64 Bits zur Verfügung. Jedes Bit steht für einen Punkt des Zeichens und läßt sich einzeln ein- oder ausschalten.

Leider wird der Charactergenerator im 40-Zeichen-Modus direkt aus dem ROM gelesen. Das bedeutet, hier gibt es keinen bereits ins RAM kopierten Zeichensatz, der leicht verändert werden könnte. Dies ist aber einzusehen, da der 40-Zeichen-Controller VIC auf normalen Adreßraum zurückgreift, um nicht wertvollen Speicherplatz zubelegen.

Aus dem oben Gesagten geht hervor, daß der Zeichensatz erst ins RAM kopiert werden muß, wenn man ihn ändern will. Da stellt sich zunächst die brennende Frage, wohin man den neuen Zeichensatz kopieren will. Es gibt da nur die Möglichkeit, den BASIC-Start heraufzusetzen. Geben Sie also bitte im Direktmode ein:

POKE 46,58:POKE 58*256,0:NEW

Eventuell im Speicher befindliche Programme werden dabei natürlich gelöscht.

Die eigentliche Kopierroutine ist jetzt nicht weiter schwer, wir formulieren sie zunächst in BASIC:

```
10 REM COPY CHARGEN $D000 -) $2000
20 BANK 14:REM CHARGEN AUSLESEN
30 FOR X=0 TO 4095
40 POKE DEC("2000")+X,PEEK(DEC("D000")+X)
50 NEXT X
60 BANK 15:REM NORMALE KONFIG.
70 END
```

Hier macht sich wieder das langsame BASIC bemerkbar. Die Ausführung dieser kleinen Routine zieht sich über eine Minute hin; haben Sie Geduld!
Der Zeichengenerator wird aus dem ROM an den BASIC-Anfang gebracht, weswegen dieser vorher unbedingt hochgesetzt werden mußte.

So, der Zeichengenerator befände sich jetzt im RAM. Das weiß der Rechner jedoch nicht; es muß ihm erst mitgeteilt werden, damit er vom ROM-Zeichensatz auf den entsprechenden RAM-Charset zurückgreift. Bewanderte C64-Fans werden in diesem Zusammenhang sicher Adresse 53272 kennen. Da der VIC kompatibel geblieben ist, hat auch diese Adresse ihre Bedeutung nicht verloren. Der Inhalt der Speicherstelle

bestimmt nämlich die Anfangsadresse vom Charactergenerator und vom Bildschirm-RAM. Das letztere wollen wir hier zunächst vernachlässigen; es geht uns erst einmal nur um den Zeichengenerator. Hier eine Übersicht der möglichen Startkombinationen:

Bildschirmspeicher		Zeichensatz	
0000xxxx	0	xxxx000x	0
0001xxxx	1024 (normal)	xxxx001x	2048
0010xxxx	2048	xxxx010x	4096 (normal)
0011xxxx	3072	xxxx011x	6144
0100xxxx	4096	xxxx100x	8192
0101xxxx	5120	xxxx101x	10240
0110xxxx	6144	xxxx110x	12288
0111xxxx	7168	xxxx111x	14336
1000xxxx	8192		
1001xxxx	9216		
1010xxxx	10240		
1011xxxx	11264		
1100xxxx	12288		
1101xxxx	13312		
1110xxxx	14336		
1111xxxx	15360		

Der Zeichensatz läßt sich also in 2k-Schritten im Speicher verschieben. Zwei Dinge fallen auf: Erstens kann der Zeichensatz nur innerhalb der ersten 16k des Speichers verschoben werden. Zweitens befindet sich der normale ROM-Zeichensatz scheinbar ab Adresse 6144. Doch dort ist normalerweise BASIC-RAM. Wie ist das zu erklären?

Zunächst einmal kann der VIC nur 16k Speicher adressieren. In diesem Fall handelt es sich um die ersten 16k im Speicher. Die Adresse 6144, auf die er zugreift, steht also nur für die Adresse 6144 innerhalb dieses 16k-Blockes.
Welcher 16k-Block eingeschaltet ist, erfahren wir aus dem Inhalt der Adresse 56576:

16-K-Bereich:

$0000 - $3FFF	0 - 16383	POKE 56576,199
$4000 - $7FFF	16384 - 32767	POKE 56576,198
$8000 - $BFFF	32768 - 49151	POKE 56576,197
$C000 - $FFFF	49152 - 65535	POKE 56576,196

Da normalerweise der letzte 16k-Block eingeschaltet ist, beginnt der Zeichengenerator also bei Adresse 49152 + 4096 = 53248 ($D000). Wenn Sie jetzt noch einmal die erste Tabelle aufschlagen, sehen Sie, daß bei der Adresse $D000 der normale Zeichensatz (Großbuchstaben und Grafikzeichen) beginnt.
Zusammenfassend kann man sagen, daß die ersten beiden Bits der Speicherstelle 56576 die Adreßbits 14 & 15 des Zeichengenerators darstellen.

Bitte beachten Sie, daß auch der Bildschirmspeicher in 16k Schritten verschoben wird. Es muß also auch das Hi-Byte des Bildschirm-RAMs verändert werden:

```
POKE 2619,4+X*64
```

X=0-3 (entsprechend 16k-Bank 0-3)

Um dem Rechner mitzuteilen, daß sich der neue Zeichensatz ab $2000 im Speicher befindet, muß also der Inhalt der Speicherstelle 53272 verändert werden. Und hier gibt es einen ganz wesentlichen Unterschied gegenüber dem C64. Während man dort einfach durch POKE den Inhalt dieses Registers verändern konnte, geht dies hier nicht mehr. Egal, was Sie in diese Adresse poken, der Inhalt wird immer neu gesetzt.
Vielmehr bedient man sich eines Bytes in der Zeropage:

$0A2C (2604) VIC TEXT SCREEN/CHAR BASE POINTER

Die Sache erscheint komplizierter als sie ist. Haben Sie beim C64 Werte in Adresse 53272 gepoked, so tun Sie jetzt dasselbe mit Adresse 2604. Der Inhalt dieser Adresse wird dann automatisch in Adresse 53272 geschrieben.

Die Umschaltung auf unseren neuen Zeichensatz sieht dann so aus:

```
POKE 2604,PEEK(2604) AND NOT 2+4+8 OR 8
```

Zur Erklärung: Es werden zuerst die Bits 1 - 3 (die für die Lage des Zeichengenerators zuständig sind) gelöscht und dann das Bit 3 gesetzt. Dadurch ergibt sich für die Adresse 53272 folgender Inhalt: xxxx100x

Nach Eingabe dieser Zeile beginnt der neue Zeichengenerator ab Adresse 8192. Dies natürlich auch, wenn Sie die eingangs beschriebene, etwas langwierige BASIC-Kopierroutine nicht benutzt haben; dann jedoch sehen die Zeichen auf dem Schirm etwas merkwürdig aus.

1.3.1 Verändern des Zeichensatzes

Geben Sie nun das folgende kleine BASIC-Programm ein:

```
10 REM Klammeraffe als Quadrat
20 FOR X=0 TO 7: REM 8 BYTES PRO ZEICHEN
30 READ CO
40 POKE 8192+0*8+X,CO
50 NEXT X
60 DATA 255,129,129,129,129,129,129,255
```

Der Klammeraffe verwandelt sich vor Ihren Augen in ein Quadrat (nähere Infos hierzu siehe "Definition des eigenen Zeichensatzes", 80-Zeichen-Bildschirm).

1.3.2 Zeichen-Editor

Auch Sie als Benutzer des 40-Zeichen-Schirms sollen Ihre neuen Zeichensätze nicht von Hand berechnen müssen: Sie können den Zeichen-Editor für den 80-Zeichen-Schirm mit geringfügigen Änderungen komplett übernehmen. Ändern Sie bitte folgende Zeilen:

```
5005 - 5045    :  fallen weg
5065 AD=8192+8*A+Y
5070 POKE AD,W
5075           :  fällt weg
```

1.4. Verschieben des Bildschirmspeichers

Der normale Bildschirmspeicher des VIC liegt im RAM von $0400 bis $07FF (1024 - 2023). Dieser läßt sich aber frei im Speicher verschieben. Wichtig sind dabei die Adressen:

```
2604  VIC TEXT SCREEN/CHAR BASE POINTER
2619  VIC TEXT SCREEN BASE
```

Der Bildschirmspeicher läßt sich in 1k-Byte großen Schritten im gesamten Speicher verschieben. Die möglichen Anfangsadressen entnehmen Sie bitte der Tabelle der Adresse 53272 im vorangegangenen Kapitel.

Wir wollen Ihnen nun eine nützliche Anwendung dieser Verschiebetechnik präsentieren:

1.4.1 Arbeiten mit mehreren Bildschirmen

Auch der Benutzer des 40-Zeichen-Bildschirms hat die Möglichkeit, mit mehreren Bildschirmen zu arbeiten. Bedienen Sie sich dazu folgenden Programms:

```
1300 78        SEI          : Interrupt verhindern
1301 A9 18     LDA #$18     : Low-Byte neuer IRQ
1303 8D 14 03  STA $0314    : und abspeichern
1306 A9 1A     LDA #$13     : High-Byte neuer IRQ
1308 80 15 03  STA $0315    : und abspeichern
130B A9 00     LDA #$00     : Länge der Funktionstasten
130D 80 00 10  STA $1000    : F1 aus
1310 8D 02 10  STA $1002    : F3 aus
1313 80 04 10  STA $1004    : F5 aus
1316 58        CLI          : Interrupt wieder erlauben
1317 60        RTS          : Zurück zu BASIC

Neuer IRQ:
```

```
1318 48        PHA          : Akku retten
1319 8A        TXA          : X-Register retten
131A 48        PHA          : auf Stapel
131B A6 D5     LDX $D5      : Nummer der gedrückten Taste
131D E0 04     CPX #$04     : F1?
131F D0 0D     BNE $132E    : nein, also nächste Abfrage
1321 A9 14     LDA #$14     : Neue Startadresse
1323 8D 2C 0A  STA $032C    : und setzen
1326 A9 04     LDA #$04     : Neuer BS bei $0400
1328 8D 3B 0A  STA $033B    : und setzen
132B 4C 50 1A  JMP $1350    : Ende F1
132E E0 05     CPX #$05     : F3?
1330 D0 0D     BNE $133F    : nein, also nächste Abfrage
1332 A9 84     LDA #$84     : Neue Startadresse
1334 8D 2C 0A  STA $0A2C    : und setzen
1337 A9 20     LDA #$20     : Neuer BS bei $2000
1339 8D 38 0A  STA #$0A3B   : und setzen
133C 4C 50 1A  JMP $1350    : Ende F3
133F E0 06     CPX #$06     : F5?
1341 D0 0D     BNE $1350    : nein, also fertig
1343 A9 94     LDA #$94     : Neue Startadresse
1345 8D 2C 0A  STA #$0A2C   : und setzen
1348 A9 24     LDA #$24     : Neuer BS bei $2400
134A 8D 3B 0A  STA $0A3B    : und setzen
134D 4C 50 1A  JMP $1350    : Ende F5
1350 68        PLA          : Altes X-Register
1351 AA        TAX          : und wiedergeben
1352 68        PLA          : Akku zurückholen
1353 4C 65 FA  JMP $FA65    : zur normalen IRQ-Routine
```

Die Routine arbeitet nach dem gleichen Prinzip wie die für den 80-Zeichen-Bildschirm.

Sobald das Maschinenprogramm gestartet worden ist, können Sie durch Tastendruck (F1, F3 und F5) zwischen drei unabhängigen Bildschirmseiten hin- und herschalten. Anfangs sind diese gewöhnlich voller merkwürdiger Zeichen. Löschen Sie den Bildschirm dann einfach durch:

PRINT CHR$(147) (oder auch durch CLR / HOME)

Im Programm können Sie zwischen den Bildschirmen so umschalten:

```
POKE 213,4   (normaler BS)
POKE 213,5   (2. BS bei $2000)
POKE 213,6   (3. BS bei $2400)
```

Neue Speicherkonfiguration:

	BS 1 (normal)	BS 2	BS 3
Bildschirmspeicher-Anfang	$0400	$2000	$2400
Bildschirmspeicher-Ende	$07FF	$23FF	$27FF
Farb-RAM-Anfang	$0800	$0800	$D800
Farb-RAM-Ende	$DBFF	$DBFF	$DBFF
BASIC-Start	$2800	$2800	$2800
Konfiguration nach:	F1	F3	F5

1.5. Der 80-Zeichen-Bildschirm

Da der C-128 nicht nur für Spiel-, sondern auch für Anwendungszwecke gedacht ist, wurde ein Grafikprozessor, der 80 Zeichen pro Zeile darstellen kann, mit eingebaut.
Dieser Prozessor trägt die Bezeichnung VDC 8563 (VDC steht für "video display controller").
Was dieser Controller vermag, kann man ruhigen Gewissens als kleine Sensation bezeichnen.

Zunächst sei hier nochmals angemerkt, daß die Darstellung von 80 Zeichen pro Zeile über diesen Chip nur mit einem RGB-Monitor funktioniert.
Da jedoch beide Darstellungsmodi - 40 und 80 Zeichen pro Zeile - über einen eigenen Controller verfügen, ist auch die gleichzeitige Darstellung auf zwei Bildschirmen möglich. Darauf werden wir an späterer Stelle eingehen.

Die erste Frage von passionierten Programmierern wird wohl sein: Wo liegen...

1.6. Bildschirm- und Farb-RAM

Und hier sieht sich der Anwender etwas völlig Neuem gegenüber. Während beim 40-Zeichen-Bildschirm Farb- und Bildschirmspeicher im normalen RAM lagen, also leicht mit PEEK und POKE zu manipulieren waren, liegt das 80-Zeichen-Video-RAM außerhalb des normalen Adreßbereiches und kann nicht durch (normale) PEEK- oder POKE-Funktionen verändert werden! Kein Grund zur Panik. Sie haben ja noch uns. Wir werden Ihnen auf den nächsten Seiten zeigen, wie ein Zugriff auf das Video-RAM dennoch möglich ist. Dazu ist es jedoch wichtig, wenigstens teilweise die Register des VDC 8563 zu kennen. Deshalb finden Sie hier eine kurze Registerbeschreibung dieses Chips. Sicherlich wird sie Ihnen jetzt noch nicht viel sagen. Auf den folgenden Seiten werden wir die Funktion einiger Register jedoch an vielen Beispielen erklären. Sie werden dann sicher die vielen Vorteile des neuen Chips zu schätzen wissen!

1.7. Registerbeschreibung des 80-Zeichen-Controllers

```
REG Bedeutung
*************************************************************
00   READ:  Status, LP, VBlank,-,-,-,-,-
     WRITE: Bit 0-5 Gewünschtes Register
01   Zeichen pro Zeile
02   Verschieben des BS-Fensters (horizontal, zeichenweise)
03   Verschieben des BS-Fensters (horizontal, pixelweise)
04   Synchronisation vertikal
```

05 Vertikal Total
06 Zeilen pro Bildschirmseite
07 Verschieben des BS-Fensters (vertikal, zeilenweise)
08 Interface-Mode
09 Matrix-Register vertikal
10 Cursor Mode, Beginn Scan
11 Ende Scan
12 Bildschirmspeicher Anfangsadresse HI
13 LO von 12
14 Cursor Position HI
15 LO von 14
16 Light Pen vertikal
17 Light Pen horizontal
18 Kanaladresse HI
19 LO von 18
20 Attribut-RAM Startadresse HI
21 LO von 20
22 Matrix-Register horizontal, Matrix Display horizontal
23 Matrix Display vertikal
24 Soft-Scroll vertikal
25 Soft-Scroll horizontal
26 Farbe
27 AdreßVerschiebung
28 Charactergenerator Basis-Adresse HI
29 Unterstreich-Cursor-Scan-Linie
30 Wiederholungsregister
31 Kanal, Byte read/write in Video-RAM
32 Block Startadresse HI
33 LO von 32
34 Beginn der Bildschirmdarstellung
35 Ende der Bildschirmdarstellung
36 Refresh-Rate

1.8. Der Display-Controller

Die Bedeutung der 37 Register dieses Controllers dürften nun einigermaßen bekannt sein. Wir werden mit den folgenden Beispielen einige der Register verändern. Daher stellt sich

jetzt die Frage, wie Registerinhalte verändert werden können.
Die Register des Controllers werden indirekt adressiert. Das bedeutet, lediglich die Register 0 und 1 liegen im Speicher. Alle anderen Register werden über diese beiden adressiert. Wollen Sie beispielsweise den Inhalt des Registers 26 erfragen, funktioniert dies so:

```
A=DEC("0600")
POKE A,26:PRINT PEEK (A+1)
```

In Register 0 ($D600) wird die Nummer des Registers geschrieben, mit dem kommuniziert werden soll. Anschließend bildet Register 1 ($D601) den Kanal zum gewünschten Register und kann beschrieben oder ausgelesen werden:

```
10 INPUT "REGISTER";R
20 INPUT "WERT";W
30 POKE DEC("D600"),R
40 POKE DEC("D601"),W
```

Zugriff auf das Video-RAM

Wie bereits erwähnt, belegt das Video-RAM einen 16k großen Speicherbereich außerhalb des normalen Adreßbereiches. Dies ist vorteilhaft, da so dem normalen Adreßraum kein Speicherplatz geklaut wird, der für BASIC-Programme dringend benötigt werden könnte. Möchte man auf das Video-RAM zurückgreifen, bedient man sich von BASIC aus am einfachsten folgender Methode:

```
100 BA=DEC("D600")
110 INPUT "ADRESSE DES VIDEO-RAMs";V
120 INPUT "WERT";W
130 HI=INT(V/256):LO=V-(256*HI)
140 POKE BA,18:POKE BA+1,HI
150 POKE BA,19:POKE BA+1,LO
160 POKE BA,31:POKE BA+1,W
```

```
170 WAIT BA,32
180 POKE BA,30:POKE BA+1,1
```

Programmerklärung:

100 : Basisadresse des VDC in BA speichern
110 : Gewünschte Adresse des Video-RAMs erfragen; V
120 : Wert erfragen, der in diese Adresse geschrieben werden soll; W
130 : Zerlegen der Adresse V in Low- und High-Byte
140 : High-Byte der gewünschten Adresse in Register 18
150 : Low-Byte der gewünschten Adresse in Register 19
160 : Byte-Wert in Register 31
170 : Ein Zugriff auf das Video-RAM kann von BASIC aus nur während des vertikalen Strahlenrücklaufes erfolgen. Auf diesen wird hier gewartet.
180 : Der Zeichenzähler (Register 30) wird mit 1 gefüllt (= ein Zeichen ausgeben).

Nun zur Belegung des Video-RAMs nach dem Einschalten des Rechners:

$0000 - $07FF Bildwiederholungsspeicher (BS-RAM)
(dez. 0-2047)

$0800 - $0FFF Attribut-RAM (u.a. Farbspeicher)
(dez. 2048-4095)

$1000 - $1FFF frei
(dez. 4096-8191)

$2000 - $3FFF Zeichengenerator
(dez. 8192-16385)

Jetzt wollen wir doch einmal unsere Routine zur Beschreibung des Video-RAMs ausprobieren: Starten Sie das obige BASIC-Programm und geben Sie als gewünschte Adresse 0 ein. Den Wert wählen Sie frei zwischen 0 und 255.

Jetzt müßte in der linken oberen Ecke des 80-Zeichen-Bildschirmes ein Zeichen erscheinen. Welches, hängt von Ihrem Wert ab.
Wir möchten Ihnen nicht verschweigen, daß diese Art des Video-RAM-Zugriffs von BASIC aus sehr unzuverlässig ist. Es kann schon einmal vorkommen, daß in Zeile 110 der vertikale Strahlenrücklauf erkannt wird, wenn er fast zu Ende ist. Der nachfolgende Schreibzugriff erfolgt dann nicht mehr in dieser Phase und wird wirkungslos. Sie können zwar versuchen, mehrmals in eine Speicherstelle zu schreiben (einmal klappt es bestimmt!), aber diese Art des Zugriffs ist wirklich nichts für eigene Programme. Sie ist hier nur der Vollständigkeit halber erwähnt.

1.9. Zuverlässiger Video-RAM-Zugriff

Wie macht es denn das Betriebssystem, daß immer ein Zeichen auf den 80-Zeichen-Bildschirm ausgegeben wird, wenn eine Taste gedrückt wird? Da scheint es doch perfekt zu funktionieren! Wir wollen uns deshalb einfach die Methode des Betriebssystems zunutze machen, um ein Zeichen auf den Bildschirm zu bringen. Es gibt für diese Aufgabe nämlich eine interessante ROM-Routine, die sich leicht von Ihnen mißbrauchen läßt. Das Format sieht so aus:

```
SYS 49155,CHARACTER,FARBE
```

CHARACTER=Zeichen (0-255)
FARBE=Zeichenfarbe (0-16)

Dieser Zugriff auf das Video-RAM funktioniert immer, und er hat noch einen weiteren Vorteil: Die von uns benutzte ROM-Routine gibt das Zeichen nicht ausschließlich auf den 80-Zeichen-Bildschirm aus. Ist zufällig der 40-Zeichen-Schirm eingeschaltet, dann wird das Zeichen eben auf diesen Schirm ausgegeben. Es ist Ihnen so möglich, Programme zu schreiben, die unabhängig vom gerade eingeschalteten Bildschirm Zeichen ausgeben!

Wenn Sie einmal die oben genannte Routine ausprobieren, werden Sie feststellen, daß das Zeichen immer in die folgende Zeile gebracht wurde. Das liegt daran, daß die Position, an der das Zeichen ausgegeben werden soll, aus den Speicherstellen 224 und 225 gelesen wird: der aktuellen Cursorposition. Möchten Sie ein Zeichen aber an eine beliebige Stelle des Schirms schreiben, manipulieren Sie einfach diese Speicherstellen:

```
10 AD=Spalte+PEEK(238)*Zeile
20 S1=PEEK(224):S2=PEEK(225)
30 HI=(AD/256):LO=AD-(256*HI)
40 POKE 224,LO:POKE 225,HI
50 SYS 49155,CHARACTER,FARBE
60 POKE 224,S1: POKE 225,S2
70 END
```

```
Spalte  : 0-79 (0-39)
Zeile   : 0-24

238     : Maximale Länge der Bildschirmzeile
224     : Cursorposition lo
225     : Cursorposition hi
49155   : Startadresse ROM-Routine
```

1.10. POKE-Simulation

Die folgende Maschinenroutine macht Schluß mit Kompromissen. Wir haben sie "modifizierter POKE-Befehl" genannt. Es ist quasi ein POKE-Befehl speziell für das 80-Zeichen Video-RAM. Zunächst das Assembler-Listing

```
1800  48        PHA        : Zeichen auf Stapel retten
1801  8A        TXA        : Low-Byte Adresse
1802  48        PHA        : auf Stapel retten
1803  98        TYA        : High-Byte Adresse
1804  48        PHA        : auf Stapel retten
1805  A9 02     LDA #$02
1807  8D 28 0A  STA $0A28  : Cursorflag setzen
180A  A2 12     LDX #$12   : VDC Register 18
180C  68        PLA        : High-Byte zurückholen
180D  20 1B 1C  JSR $181B  : Register setzen
1810  E8        INX        : VDC Register 19
1811  68        PLA        : Low-Byte zurückholen
1812  20 1B 1C  JSR $1818  : Register setzen
1815  A2 1F     LDX #$1F   : VDC Register 31
1817  68        PLA        : Zeichen zurückholen
1818  4C 1B 1C  JMP $181B  : Register setzen, fertig
-------------------------------------------------------
181B  8E 00 D6  STX $D600  : Register in REG 0
181E  2C 00 D6  BIT $D600  : Bit 7 gesetzt?
1821  10 FB     BPL $1C8E  : nein, also nochmal testen
1823  8D 01 D6  STA $D601  : Wert ans Register übergeben
1826  60        RTS
```

Alle Nicht-Maschinen-Programmierer finden hier natürlich einen BASIC-Loader:

(Programm 1.10.a)

Es steht Ihnen nun eine etwas veränderte POKE-Anweisung mit folgendem Format zur Verfügung:

```
SYS DEC("1800"),CHR,LO,HI
```

```
CHR=Zeichen/Byte-Wert (0-255)
LO =Lo-Byte der gewünschten Adresse
HI =Hi-Byte der gewünschten Adresse
```

Probieren wir es aus:

```
10 INPUT "ADRESSE ";AD
20 HI=INT(AD/256):LO=AD-(256*HI)
30 SYS DEC("1800"),3,LO,HI
```

Geben Sie als Adresse 0 ein. Sofort erscheint ein "C" in der linken oberen Ecke des 80-Zeichen-Bildschirmes (3=Code C). Starten Sie die Routine erneut. Geben Sie nun als Adresse 2048 ein. Das zuvor auf den Bildschirm gebrachte "C" verfärbt sich hellblau: Sie haben in das Attribut-RAM geschrieben (2048-3047).
Ein Byte des Attribut-RAMs ist folgendermaßen aufgebaut:

Bit 0	Helligkeit
Bit 1	Blau
Bit 2	Grün
Bit 3	Rot
Bit 4	Blinken
Bit 5	Unterstreichen
Bit 6	Reverse Darstellung
Bit 7	2. Zeichensatz

Die Bedeutung der ersten vier Bits dürfte klar sein: Sie definieren in Kombination eine der möglichen 16 Farben. Wird Bit 4 gesetzt, blinkt das entsprechende Zeichen hektisch auf dem Bildschirm. Fügen Sie doch mal in obiges Programm die Zeile:

```
30 SYS DEC("1800"),2↑0+2↑3+2↑4,LO,HI
```

ein und wählen Sie wieder als Adresse 2048. Das erste Zeichen des Bildschirmes blinkt hellrot!
Analog dazu läßt sich ein Zeichen auch unterstreichen oder

revers darstellen, wenn Bit 5 bzw. 6 gesetzt sind. Ein besonderes Bonbon stellt das 7. Bit dar. Sicher kennen Sie die Möglichkeit, durch gleichzeitiges Drücken der Tasten C= und SHIFT den Zeichensatz umzuschalten. Während diese Umschaltung sich beim 40-Zeichen-Bildschirm sofort bemerkbar macht, weil alle Zeichen umgeschaltet werden, ist es hier möglich, gleichzeitig sowohl Zeichen des 1. als auch des 2. Zeichensatzes auf dem Bildschirm darzustellen. Sie können also als Schrift Groß- und Kleinbuchstaben verwenden, ohne auf die Blockgrafik-Symbole verzichten zu müssen! Es stehen Ihnen somit 512 verschiedene Zeichen gleichzeitig zur Verfügung!

1.11. Der Charactergenerator

Wie Sie gerade erfahren haben, ist es also auf dem 80-Zeichen-Schirm möglich, bis zu 512 verschiedene Zeichen darzustellen. Dies ist zugegeben eine große Auswahl. Aber für verschiedene Spiele oder spezielle Programme (kyrillisches Alphabet, mathematische Sonderzeichen, etc.) reicht der hauseigene Zeichensatz nicht aus. Man muß also darangehen und die fehlenden Zeichen selbst definieren. Dies ist beim 80-Zeichen-Bildschirm zudem sehr leicht, da sich der Zeichengenerator ohnehin schon im RAM befindet ($2000-$3FFF im Video-RAM) und direkt dort verändert werden kann!

1.12. Auslesen des Zeichengenerators

Wir werden jetzt den Zeichengenerator aus dem Video-RAM auslesen. Die gelesenen Informationen werden dann wieder in Zeichen zurückverwandelt. Sehen Sie selbst:

(Programm 1.12.a)

```
Verwendete Variablen:

BA  : Basis des Zeichengenerators
```

```
A  : Basis des VDC
B  : REG 1 des VDC
AD : Aktuelle Adresse im Zeichengenerator
HI : High-Byte von AD
LO : Low-Byte von AD
CH : Ausgelesener Wert einer Buchstabenzeile
W  : Zähler
```

Nach dem Starten des Programms werden alle im Zeichengenerator gespeicherten Zeichen vergrößert auf dem Bildschirm abgebildet. Auffällig ist, daß nach jedem Zeichen acht Leerzeilen folgen. Und tatsächlich: Der Zeichengenerator des Commodore 128 (für die 80-Zeichen-Darstellung) ist anders aufgebaut als der des C-64 oder VC-20. Die folgende Zeichnung verdeutlich das:

(Zeichnung 1.12.a)

Sie sehen: Für jedes Zeichen stehen 16 Bytes an Stelle von 8 Bytes beim C-64 oder VC-20 zur Verfügung. Allerdings werden davon wie beim C-64 auch nur acht Bytes benötigt. Sollte das Verschwendung sein?! Keineswegs. Zwar sind im Normalfall jeweils acht Bytes pro Zeichen unbenutzt. Diese acht "Leerbytes" erfüllen jedoch eine bestimmte Funktion. Der neue Display-Controller VDC 8563 ist nämlich in der Lage, statt der üblichen 8*8-Punktematrix, aus der ein Zeichen normalerweise besteht, Zeichen aus einer Punktmatrix von bis zu 16*8 Punkten darzustellen - ganze Kleingrafiken also! Für diese Betriebsart werden nun aber 16 Bytes pro Zeichen benötigt. Daher die acht "Leerbytes".
Wie diese Vergrößerung der Matrix funktioniert, finden Sie auf den folgenden Seiten.

Wir möchten Ihnen hier noch ein weiteres nützliches Programm zum Thema "Auslesen des Charactergenerators" vorstellen:

1.13. Big Script mit Strings

Was hindert uns daran, das vergrößerte Aussehen eines Zeichens einem String zuzuordnen? Es wäre dann möglich, um

den Faktor 8 vergrößerte Schrift auf den Bildschirm zu bringen! Genau dies wollen wir tun. Das folgende Programm liefert die vergrößerte Matrix des in Z$ gespeicherten Zeichens in dieselbe Variable zurück:

(Programm 1.13.a)

1.14. Transparente drucken für den Heimgebrauch

Genau so, wie wir die vergrößerten Zeichen soeben auf den Bildschirm ausgegeben haben, könnte man sie doch auch auf den Drucker ausgeben! Auf diese Art und Weise lassen sich beliebig lange Transparente drucken. Die Schriftgröße ist sogar frei wählbar zwischen 8- und 80-facher Vergrößerung:

(Programm 1.14.a)

1.15. Definition des eigenen Zeichensatzes

Jetzt geht es dem "alten" Commodore-Zeichensatz an den Kragen! Wir werden nun diesen hauseigenen Zeichensatz verändern. Unsere Um-Definitions-Programme bedienen sich der modifizierten POKE-Anweisung, welche im Kapitel 1.10. beschrieben wurde. Bitte stellen Sie vor die nun folgenden Programme den dort genannten BASIC-Loader.

Jedes Zeichen besteht normalerweise aus 8*8 Punkten. Wenn Ihr Monitor eine gute Auflösung besitzt, so müßten Sie diese Punkte sogar erkennen können. Die Matrix des Buchstabens "A" könnte also so aussehen:

```
76543210
...**...0
..****..1
.**..**.2
.******.3
.**..**.4
.**..**.5
.**..**.6
........7
```

Die Matrix eines jeden Zeichens konnten Sie ja bereits mit Hilfe der Zeichengenerator-Lese-Programme begutachten. Wenn Sie nun eigene Zeichen definieren möchten, steht Ihnen dieselbe 8*8-Punktematrix zur Verfügung (wie man diese noch erweitern kann, finden Sie auf den folgenden Seiten!). Jeder Zeile des Zeichens ist ein Byte zugeordnet, jedes Byte wiederum besteht aus acht Bits. Jedes Bit stellt also einen Punkt dar, wenn es gesetzt ist.

Starten Sie einmal das folgende Programm:

```
 10 A=DEC("D600"):B=A+1
 20 BA=8192:ZE=0
 30 FOR X=0 TO 7
 40 AD=BA+X+(8*ZE)
 50 HI=INT(AD/256):LO=AD-(HI*256)
 60 READ CH
 70 SYS DEC("1800"),CH,LO,HI
 80 NEXT X
 90 END
100 DATA 255,129,129,129,129,129,129,255
```

Nach dem Starten des Programms verwandelt sich der Klammeraffe in ein Quadrat! Hier die Programmerklärung:

```
10   : A=Basis des VDC, B=Register 1
20   : BA=Basis des Zeichengenerators, ZE=zu veränderndes
       Zeichen (1=A, 2=B, 3=C, etc.)
30   : 8 Bytes verändern
40   : Adresse des Bytes=Basis des Zeichengenerators+Byte-
       Nummer+8*Zeichennummer
50   : Zerlegung der Adresse AD in Lo- und Hi-Byte
60   : Lesen des neuen Byte-Wertes
70   : modifizierte POKE-Routine "poked" den Wert ins Video-
       RAM
100  : DATAs für das Quadrat:
```

```
76543210
********0 255 (2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0)
*......*1 129 (2^7+2^0)
*......*2 129 (2^7+2^0)
*......*3 129 (2^7+2^0)
*......*4 129 (2^7+2^0)
*......*5 129 (2^7+2^0)
*......*6 129 (2^7+2^0)
********7 255 (2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0)
```

Über Sinn und Unsinn eines Quadrates läßt sich natürlich streiten (wo man damit doch so schöne Backsteinmauern printen kann!?). Probieren Sie deshalb doch mal folgende DATA-Zeilen in obigem Programm aus:

```
100 DATA 60,66,157,161,161,157,66,60
```

oder:

```
100 DATA 66,157,161,161,161,161,157,66
```

Ihnen steht so das (C)-Zeichen auf der Klammeraffen-Taste zur Verfügung!

1.16. Mit dem Zeichen-Editor: Definition leicht gemacht!

Zwar könnten Sie Ihre gesamten neuen Zeichendefinitionen per Hand und Rechenkästchen auf oben genannte Art und Weise ins Byte-Format bringen. Aber wofür hat man denn sein Arbeitstier Computer? Dieser erledigt dieselbe Aufgabe genauso zuverlässig:

(Programm 1.16.1)

```
Variablen:

ZN    : Erste Zeilennummer der DATA-Zeilen
MA    : Matrix 8*MA+1
D(x)  : berechnete DATAs für Zeichen
W$(x) : eingegebene Zeichenzeilen
C$    : veränderbares Zeichen
AC    : ASCII-Code von C$
CO    : Bildschirmcode des ASCII-Zeichens
CZ    : ausgelesene ROM-Daten
CS    : Checksumme
```

Programmbeschreibung:

Lassen Sie sich nicht von der Länge des Programms abschrecken; für einen Zeichengenerator mit diesen Möglichkeiten ist es immer noch kurz! Saven Sie das Programm nach dem Eingeben zunächst ab.

Nach dem Starten meldet sich der Generator und fragt nach dem Zeichen, das Sie verändern wollen. Drücken Sie einfach die entsprechende Taste. Nun werden zwei 8*8 Zeichen große Felder ausgegeben. Im linken Feld erscheint die Originalmatrix des gewünschten Zeichens, frisch aus dem ROM gelesen. Das rechte Feld können Sie nun selbst beschreiben. "*" bedeutet dabei Punkt gesetzt; "." entsprechend kein Punkt gesetzt.

Nach Beendigung jeder Zeile drücken Sie bitte die RETURN-Taste, um in die folgende zu gelangen. Ist das gesamte Zeichen definiert, haben Sie die Möglichkeit, nachträglich Korrekturen anzubringen. Wünschen Sie dies, drücken Sie "J" und können so die selbstdefinierte Matrix einer erneuten Veränderung unterwerfen. Nicht mehr zu verändernde Zeilen können einfach mit "RETURN" übergangen werden.
Ist das Zeichen zu Ihrer vollen Zufriedenheit ausgefallen, können Sie es übernehmen. Dabei werden die berechneten DATAs des Zeichens in eine Zeile des Programms geschrieben; der Zeichengenerator programmiert sich also selbst!
Falls Sie das Zeichen NICHT übernehmen, ist die selbstdefinierte Matrix wieder verloren und das Zeichen behält sein normales Äußeres.

Wenn alle Zeichen Ihren Wünschen entsprechend verändert sind, können Sie das Programm mit der RUN/STOP-Taste unterbrechen. Anschließend braucht der Generator nur noch durch:

DELETE -5000

gelöscht zu werden. Ihnen steht jetzt ein kompletter BASIC-Loader zur Verfügung, der mit Zeile 5000 beginnt (mit RENUMBER können Sie die Zeilennummern Ihren Vorstellungen anpassen). Der BASIC-Loader kann unverändert in Ihre Programme übernommen werden und sorgt dafür, daß (auf dem 80-Zeichen-Bildschirm) Ihre neudefinierten Zeichen ausgegeben werden. Alle übrigen Zeichen behalten ihr normales Aussehen.

Das Programm:

```
50 - 120    : Das in C$ gespeicherte Zeichen wird in den
              Commodore-spezifischen Bildschirmcode verwan-
              delt, der sich vom normalen ASCII-Code stark
              unterscheidet.
              Der Bildschirmcode ist dann in CO abgelegt.
```

140- 232 : Die Bildschirmmaske wird aufgebaut. Die Speicherkonfiguration wird mit BANK 14 umgeschaltet, damit der ROM-Zeichensatz ab $D000 ausgelesen werden kann. Anschließend wird Zeile für Zeile des Zeichens aus dem ROM gelesen und die Bits gesetzt.
Nach dem Auslesen wird wieder die normale Speicherkonfiguration durch BANK 15 eingestellt.

260- 351 : Editor-Routine: Der Bildschirm wird als Datei eröffnet, damit durch die INPUT-Anweisung kein störendes Fragezeichen ausgegeben wird.
Jede Eingabezeile der User-Matrix wird als Fenster definiert, damit diese Zeile nicht versehentlich verlassen werden kann.
In D(Y) wird die berechnete Bit-Summe jedes Zeichens gespeichert und schließlich die Bildschirmdatei wieder geschlossen.

360- 400 : Abfrage: Soll noch korrigiert werden? Wenn ja, wird MF=0 gesetzt, damit beim erneuten Durchgang die Windows nicht gelöscht werden.
D(Y) wird gelöscht, da es erneut berechnet werden muß.
Soll daß Zeichen nicht übernommen werden, wird wieder zum Programmanfang verzweigt.

1000-1039 : Die DATA-Zeile des letzten Zeichens wird ausgeprintet. Das Format: Zeichencode, 8 Daten. Dahinter folgt als Ende-Kennzeichen eine Zeile DATA -1.
Um den Rücksprung zu sichern, wird zusätzlich GOTO 30 ausgegeben; Neustart.
Die DATA-Zeilennummer ZN wird um 10 erhöht.

1040-1070 : Der Tastaturpuffer wird mit "RETURN" gefüllt und die Zeile übernommen. Wie dies im einzelnen funktioniert, entnehmen Sie bitte dem Kapitel 5: "System der 1000 Möglichkeiten"

schleife für die noch kommenden DATA-Statements.

5000-5085 : Hier steht der Kopf des künftigen Basic-Loaders. In den Zeilen 5005 bis 5045 wird die modifizierte POKE-Anweisung implementiert. Darunter findet sich die eigentliche Lese-

1.17. Arbeiten mit mehreren Bildschirmen

Schauen Sie sich doch einmal die Belegung des Video-RAMs an:

```
$0000 - $07FF  Bildwiederholungsspeicher (BS-RAM)
(dez. 0-2047)

$0800 - $0FFF  Attribut-RAM (Farbe, etc.)
(dez. 2048-4095)

$1000 - $1FFF  frei
(dez. 4096-8191)

$2000 - $3FFF  Zeichengenerator
(dez. 8192-16385)
```

Auffällig ist der Bereich $1000-$1FFF. Dieser 4k große Speicherbereich mitten im Video-RAM ist nämlich unbenutzt. Dies sollte man nicht einfach hinnehmen, sondern sich überlegen, was mit diesem wertvollen Speicherplatz sinnvollerweiseangefangen werden könnte! Nun ja, 4k sind genau 2*2k, und der Bildschirmspeicher ist 2k groß. Da liegt es auf der Hand, als mögliche Anwendung zwei weitere Bildschirmseiten in den freien Bereich zu legen. Damit stünden Ihnen dann drei volle Seiten Bildschirm zur Verfügung! Was damit anzufangen ist, ist kaum auszudenken: Zum Beispiel könnte sich auf einem noch unsichtbaren Bildschirm bereits eine Grafik aufbauen, während noch munter auf einem anderen Bildschirm mit dem Anwender geflirtet wird. Erst, wenn die Grafik vollständig erstellt ist, wird von einem Bildschirm auf den anderen umgestellt, und die Grafik erscheint blitzartig! Oder auf einem Bildschirm befindet sich das Listing, auf dem zweiten das Inhaltsverzeichnis der Disk und auf dem dritten wird das Programm erstellt. So kann nach Bedarf einfach umgeschaltet werden, wenn die entsprechenden Informationen benötigt werden.

Realisierung:

Normalerweise befindet sich der Anfang des Bildwiederholspeichers ab $0000 im Video-RAM. Dieser Platz ist aber nicht bis in alle Ewigkeit bindend. Register 12 und 13 des VDC enthalten nämlich High- und Low-Byte der Anfangsadresse des Bildschirmspeichers. Hier kann selbstverständlich manipuliert werden. Insgesamt müssen drei Adressen verändert werden, wenn der Anfang des Bildschirmspeichers verschoben werden soll:

```
Register 12 des VDC:  Hi-Byte der neuen Anfangsadresse
Register 13 des VDC:  Lo-Byte der neuen Anfangsadresse
Adresse 2606($0A2E):  Hi-Byte der neuen Anfangsadresse
```

Wir wollen probeweise den Bildschirmspeicher-Anfang von Adresse $0000 nach Adresse $1000 verschieben. Das High-Byte dieser Adresse ist 16 (1*4096/256):

```
10 A=DEC("D600"): B=A+1
20 HI=16: LO=0
30 POKE A,12: POKE B,HI
40 POKE A,13: POKE B,LO
50 POKE 2606,HI
```

Sie sehen jetzt ein wüstes Durcheinander von Zeichen auf dem Bildschirm. Das ist normal, da dieser Speicherbereich normalerweise vor sich hin verstaubte und zufällige Werte enthielt. Löschen Sie den Bildschirminhalt einfach mit:

```
PRINT CHR$(147);
```

Sie können diesen Bildschirm wie den normalen benutzen. Wollen Sie wieder zu Ihrem alten Bildschirm zurück, so ersetzen Sie Zeile 20 durch:

```
20 HI=0: LO=0
```

Sie sehen, dort ist alles beim Alten geblieben und Sie können weitermachen, wo Sie aufgehört haben (mitunter ist der Cursor nicht zu sehen, sobald Sie aber ein Zeichen auf den Bildschirm tippen, blinkt er wieder in alter Frische).

Das folgende Maschinenprogramm nutzt den gesamten freien Speicherplatz. Es ist in den Interrupt eingebunden und nach dem Starten sofort aktiv. Zunächst das Assembler-Listing:

Initialisierung:

```
1C00 78        SEI        : Interrupt ignorieren
1C01 A9 1C     LDA #$1C
1C03 8D 14 03  STA $0314  : Interrupt-Zeiger verbiegen (lo)
1C06 8D 15 03  STA $0315  : Interrupt-Zeiger verbiegen (hi)
1C09 58        CLI        : Interrupt wieder zulassen
1C0A A9 00     LDA #$00
1C0C 8D 00 10  STA $1000  : F1 löschen
1C0F 8D 02 10  STA $1002  : F3 löschen
1C12 8D 04 10  STA $1004  : F5 löschen
1C15 60        RTS        : fertig
```

Neuer Interrupt:

```
1C1C 48        PHA        : Akku retten
1C1D 8A        TXA
1C1E 48        PHA        : X-Register retten
1C1F A5 D5     LDA $05    : Tastaturabfrage
1C21 C9 58     CMP #$58   : keine Taste?
1C23 F0 3A     BEQ $1C5F  : dann zur normalen IRQ-Routine
1C25 A2 0C     LDX #$0C   : VDC Register 12
1C27 C9 04     CMP #$04   : F1?
1C29 F0 00     BEQ $1C38  : ja, nach $1C38
1C2B C9 05     CMP #$05   : F3?
1C2D F0 0E     BEQ $1C3D  : ja, nach $1C38
1C2F C9 06     CMP #$06   : F5?
1C31 00 2C     BNE $1c5F  : nein, zur normalen IRQ-Routine
1C33 A9 00     LDA #$00   : BS $0000
1C35 4C 42 1C  JMP $1C42  : Register setzen
1C38 A9 10     LDA #$10   : BS $1000
1C3A 4C 42 1C  JMP $1C42  : Register setzen
1C3D A9 18     LDA #$18   : BS $1800
1C3F 4C 42 1C  JMP $1C42  : Register setzen
1C42 8E 00 D6  STX $D600  : gewünschtes Register in REG 0
```

```
1C45 2C 00 D6 BIT $D600 : Bit 7 gesetzt?
1C48 10 FB    BPL $1C45 : warten
1C4A 80 01 D6 STA $0601 : Wert ins Video-RAM schreiben,REG1
1C4D 8D 2E 0A STA $0A2E : Zeiger in Zeropage setzen
1C50 A2 0D    LDX #$0D  : VDC Register 13
1C52 A9 00    LDA #$00  : Lo-Byte =0
1C54 8E 00 D6 STX $0600 : Register setzen
1C57 2C 00 D6 BIT $D600 : Bit 7 gesetzt?
1C5A 10 FB    BPL $1C57 : warten
1C5C 80 01 D6 STA $D601 : Byte ins Video-RAM schreiben,REG1
1C5F 68       PLA       : X-Register zurück
1C60 AA       TAX
1C61 68       PLA       : Akku zurückholen
1C62 4C 65 FA JMP $FA65 : weiter zur IRQ-Routine
```

Die BASIC-Programmierer finden wieder einen Loader:

(Programm 1.17.a)

Nach dem Starten des Initialisierungsprogramms wird der IRQ-Vektor auf den zweiten Teil der Routine "gebogen". Diese Routine wird also in den IRQ eingebunden. Sie wird nun vom Rechner automatisch ca. alle 1/60 Sekunden ausgeführt.

Sobald das Maschinenprogramm gestartet ist, stehen Ihnen 3 voneinander weitgehend unabhängige Bildschirmseiten zur Verfügung. Die Umschaltung der Seiten kann auf zweierlei Weise vorgenommen werden:

a) Im Direktmode:
   Die Tasten F1, F3 und F5 schalten bei Betätigung auf die jeweilige Seite um. Jede der neuen Seiten sollte vor der Benutzung gelöscht werden.
b) Im Programm:
   Man simuliert einfach die Betätigung der Funktionstasten auf folgende Art und Weise:

```
POKE 213, 4    (Normaler BS ein)
POKE 213, 5    (2. BS ein)
POKE 213, 6    (3. BS ein)
```

So dürfte die Programmierung hochkarätiger Software nicht mehr schwer sein. Durch die Umschaltungsmöglichkeit ergibt sich eine bisher nicht gekannte Anwendungsfreundlichkeit, die Sie sich zunutze machen (Listing auf BS1, Programmieren auf BS2), um Ihren eigenen Programmen die besondere Note zu geben!

1.18. Sinnvolle Manipulationen des VDC 8563

Eingangs erwähnten wir die Vielzahl von Möglichkeiten des neuen 80-Zeichen-Display-Controllers. Um Ihnen zu zeigen, daß wir nicht zuviel versprochen haben, wollen wir Ihnen nun an Hand einiger Beispielprogramme zeigen, was Sie mit ihm alles tun können. Wohlgemerkt: Wir wollen Ihnen nur Anregungen geben. Wir glauben aber, daß Sie viele dieser Beispielprogramme auch unverändert zur Bereicherung Ihrer eigenen Programme übernehmen können.

Zu Anfang des Kapitels über den 80-Zeichen-Bildschirm finden Sie die vollständige Registerbeschreibung des VDC, die wir hier nur unvollständig behandeln wollen.

Wenn wir jetzt den Display Controller manipulieren, wird damit unter Umständen die gesamte normale Bildschirmdarstellung über den Haufen geworfen. Bei eigenen Experimenten mit den Registern des Chips beachten Sie bitte, daß eine vollständige Rücksetzung des Controllers in den Ausgangszustand oftmals nur durch das kurzzeitiges Ausschalten des Rechners möglich ist. Eine dauerhafte Beeinträchtigung des Rechners ist hingegen nicht zu befürchten.

Verschieben des Bildschirmfensters

Viele Besitzer des VC-20 werden die Möglichkeit kennen, das gesamte Bildschirmfenster zu verschieben. Dieser Effekt läßt sich auch mit dem VDC erreichen. Zuständig hierfür sind die

Register 2 und 7:

```
02  Verschieben des BS-Fensters (horizontal, zeichenweise)
07  Verschieben des BS-Fensters (vertikal, zeilenweise)
```

Wir werden das ausprobieren:

```
10 REM VERSCHIEBEN DES BS-FENSTERS
20 A=DEC("D600"):B=A+1
30 FOR X=0 TO 255
40 POKE A,2:POKE B,X
50 POKE A,7:POKE B,X
60 NEXT X
70 END
```

Dieses Programm läßt den Bildschirm diagonal wandern. In die Normaldarstellung gelangen Sie wieder durch RUN STOP + RESTORE.
Eine weitere Anwendung ist die...

...Explosionssimulation

Die Qualität eines Spiels lebt von der realistischen Darstellung. Außerdem machen Effekte, die nicht jeder BASIC-Programmierer kennt, ein Programm noch wertvoller. Eine "plastische" Darstellung einer Explosion könnte beispielsweise so aussehen:

(Programm 1.18.a)

Es liegt an Ihnen, zum vibrierenden Bildschirm die entsprechenden Töne zu kreieren!

Ihnen werden sicher noch eine Menge Anwendungsmöglichkeiten einfallen. Auf jeden Fall läßt sich hier eine ganze Menge machen!

1.19. Manipulation des Bildschirmformats

Gewöhnlich stehen Ihnen im 80-Zeichen-Mode, wie der Name schon sagt, 80 Zeichen pro Zeile und 25 Zeilen pro Seite zur Verfügung. Es können also insgesamt 80*25=2000 Zeichen auf dem Bildschirm dargestellt werden. Wir möchten dieses auf die Dauer doch etwas triste Format abändern. Das ist gar nicht so schwer, gibt es doch die VDC-Register 1 und 6:

```
01  Zeichen pro Zeile    (normalerweise 80)
06  Zeilen pro BS-Seite (normalerweise 25)
```

Es sollte nicht versucht werden, mehr als 80 Zeichen pro Zeile darzustellen. Statt dessen ist unser Ziel, das Bildschirmfenster nach unten zu erweitern. Hierbei ist zu beachten, daß:

(neue Anzahl Spalten)*(neue Anzahl Zeilen)

kleiner oder höchstens gleich 2000 ist, da es sonst zu Problemen bei der Darstellung kommt. Soll das Fenster also nach unten erweitert werden, geht dies auf Kosten der Zeichen pro Zeile:

30 Zeilen * 62 Zeichen

```
10 REM NEUES BILDSCHIRMFORMAT 62*30
20 A=DEC("D600"):B=A+1
30 POKE A,1:POKE B,62
40 POKE A,6:POKE B,30
50 END
```

Dieses kurze Programm stutzt das Bildschirmformat auf 30 Zeilen zu 62 Zeichen zurecht. Im Prinzip können Sie aber auch jedes andere Format definieren, wenn Sie oben Gesagtes beachten. So kann das Bildschirmformat immer an die gerade vorliegende Spielsituation angepaßt werden: ein dünner Brunnen, ein breiter Schacht etc. Eine Spielfigur könnte also direkt vom obersten Rand des Monitors bis zum unteren klettern, ohne im normalerweise vorhandenen Bildschirmrahmen zu verschwinden!

Es bietet sich Ihnen außerdem die Möglichkeit, das veränderte Bildschirmfenster Ihren Wünschen entsprechend auf der Mattscheibe zu verschieben. Wie das geschieht, entnehmen Sie bitte dem vorangegangenen Kapitel "Verschieben des Bildschirmfensters"!

1.20. So geht's effektiver für Grünmonitor-Besitzer

Wenn Sie glücklicher Besitzer eines Grünmonitors sind, werden Sie höchstwahrscheinlich nicht in den Genuß der Farbdarstellung des Rechners kommen. Allenfalls an ein paar verschiedenen GrÜnwerten läßt sich die Existenz der Farben erahnen. Wofür benötigen Sie dann eigentlich das 2k große Attribut-RAM? Diese 2k lassen sich in diesem Fall bestimmt besser als weitere Bildschirmseite verwenden, als nutzlos in der Ecke zu verstauben. Das Problem ist also, dem Rechner mitzuteilen, daß das Attribut-RAM ein Mann zuviel im Zug ist. Diese Mitteilung geht direkt an Register 25 des VDC:

25 Soft Scroll horizontal

...lautet die offizielle Bezeichnung. Hierfür sind allerdings nur die ersten vier Bits zuständig. Bit 6 entscheidet, ob das Attribut-RAM arbeitet oder nicht:

```
10 REM INAKTIVIEREN DES ATTRIBUT-RAMS
20 A=DEC("D600"):B=A+1
30 POKE A,25:POKE B,PEEK(B) AND NOT 64
40 END
```

Durch diese kleine Routine wird das Attribut-RAM ausgeschaltet. Der Speicherbereich $0800 - $0FFF im Video-RAM steht jetzt zu Ihrer freien Verfügung und ließe sich bestens für ein weiteres Bildschirmfenster nutzen.
Natürlich ist diese Umstellung auch für Besitzer eines RGB-Monitors interessant, sofern Sie auf die farbige Darstellung verzichten wollen.

Der Vorhang fällt...!

Normalerweise wird das Bildschirmfenster von einem feststehenden Rahmen umschlossen. Dieser Rahmen läßt sich mit den VDC-Registern 34 und 35 manipulieren:

```
34  Beginn der Bildschirmdarstellung
35  Ende der Bildschirmdarstellung
```

Wir wollen uns nicht mit langen Erklärungen aufhalten, sondern Ihnen die Wirkungsweise an einem kleinen Beispielprogramm verdeutlichen:

(Programm 1.20.a)

Der linke und rechte Bildschirmrand lassen sich also hin- und herverschieben, ohne den Bildschirminhalt zu verändern. So läßt sich beispielsweise ein wandernder Streifen realisieren. Wie wäre es denn mit einem Vorhang nach jeder Vorstellung? Kein Problem:

(Programm 1.20.b)

1.21. Der erste Schritt zur neuen Matrix

Bei der Definition neuer Zeichen haben Sie sie bereits kennengelernt: Die "Matrix" von 8*8 Punkten, mit der das charakteristische Äußere jedes Zeichens festgelegt ist:

```
76543210
........0
........1
........2
........3
........4
........5
........6
........7
```

So sieht die Grundmatrix normalerweise aus; jedes Pünktchen entspricht einem möglichen Punkt des Zeichens.
Sofern Sie sich bereits im vorangegangenen Kapitel mit der Definition eines eigenen Zeichensatzes beschäftigt haben, werden Sie je nach Verwendungszweck des neudefinierten Zeichens vielleicht über diese 8*8 Punktematrix geschimpft haben: Mal ist sie zu klein für die Darstellung eines Raumschiffes, mal zu groß für ein kleines Symbol. Auf den nächsten Seiten möchten wir Ihnen deshalb zeigen, wie Sie das Format der Punktematrix verändern können!

Relative Veränderung der Punktematrix

Hier wollen wir auf die in diesem Zusammenhang sehr wichtigen Register 22 und 23 eingehen:

```
22  Matrix Display horizontal
23  Matrix Display vertikal
```

Diese beiden Register entscheiden, wieviel Pixelzeilen eines Zeichens auf dem Bildschirm dargestellt werden sollen. Dies sind normalerweise sowohl horizontal als auch vertikal acht. Zuständig sind hierfür die ersten vier Bits des Registers 22 und die ersten fünf des Registers 23. Probieren wir einfach die Wirkungsweise aus:

(Programm 1.21.a)

Hiermit läßt sich übrigens auch so eine Menge machen. Zum Beispiel zur reizvollen Gestaltung einer Spielbeschreibung:

(Programm 1.21.b)

Wie die Überschrift dieses Kapitels aber bereits andeutete, handelt es sich hierbei "nur" um eine relative Veränderung der Punktmatrix; wird die Matrix verkleinert (und nur dies ist mit diesen beiden Registern zunächst möglich), bleibt eben der unbenutzte Rest der 8*8-Matrix unbenutzt. Das kann nicht der Sinn der Übung sein.

1.22. Echte Manipulation der Punktematrix

Wir wollen jetzt versuchen, eine echte 16*8-Punktematrix zu erzeugen. Das bedeutet, für jedes Bildschirmzeichen stehen 16 Pixelzeilen zu 8 Punkten zur Verfügung. Mit dieser Matrixgröße, die zwei übereinanderstehenden Zeichen entspricht, lassen sich schon ganze Kleingrafiken realisieren! Im vorangegangenen Kapitel hatten wir noch Schwierigkeiten, die ursprüngliche 8*8-Matrix zu verändern, weil die dort beschriebenen Register einfach nur Teile der Matrix "verschluckten". Sehen Sie sich einmal diese beiden Register an:

```
04  Synchronisation vertikal
09  Matrix-Register vertikal
```

Register 9 entscheidet, wieviel Pixelzeilen zu einem Zeichen gehören. Sein Inhalt muß also verdoppelt werden, denn an Stelle von 8 soll ein Zeichen nun aus 16 Zeilen bestehen. Wird der Inhalt des Registers 9 jedoch verändert, muß die vertikale Synchronisation an die veränderten Umstände angepaßt werden, da sonst das Bild flackert oder läuft. Dies erledigt Register 4. Starten Sie doch einmal folgendes kleine Programm:

```
 10 REM 16*8 PUNKTE MATRIX
 20 A=DEC("D600"):B=A+1
 30 POKE 228,16
 40 READ X
 50 IF X=-1 THEN END
 60 READ Y
 70 POKE A,X:POKE B,Y
 80 GOTO 20
 90 :
100 DATA 9,15
110 DATA 6,17
120 DATA 23,15
130 DATA 4,19
140 DATA 7,19
150 DATA -1
```

Nach dem Starten bietet sich Ihnen ein "komisches" Bild: Zwischen den Bildschirmzeilen klafft eine große Lücke, ansonsten sind aber alle Zeichen auf dem Bildschirm gut zu sehen.
Die "Lücke" erklärt sich ganz einfach so: Jedes Zeichen besteht nach der Umschaltung aus 16*8 Punkten. Die ersten acht Pixelzeilen sind normal geblieben, die folgenden 8 sind Leerzeichen, da die entsprechenden Bytes im Zeichengenerator 0 enthalten (siehe Kapitel 1.12.)

Geben Sie einmal im Direktmode ein:

```
PRINT CHR$(27)+"R"
```

Der Bildschirminhalt wird invertiert. Sie können so das Ausmaß des gesamten Bildschirms erkennen; der Bildschirminhalt ist bedeutend größer geworden. Und tatsächlich, wen es interessiert: Es stehen Ihnen hier 17 Zeilen zu 80 Zeichen zur Verfügung, pro Zeichen 16*8 Pixel. Das ergibt eine absolute Auflösung von:

```
PRINT 17*80*(16*8)
```

174.080 einzelnen Punkten !!!

Da der Speicher des Video-RAMs jedoch nur 16k umfaßt, können diese Punkte nicht unabhängig voneinander gesetzt werden, was aber nicht weiter stören muß.

```
Programmerläuterung:

10  : A=Basis VDC, B=REG 01
20  : Die untere Fenstergrenze wird gesetzt. So ist es un-
      möglich, mit dem Cursor aus dem Bild zu verschwinden.
30  : Die VDC-Register werden mit den neuen Werten geladen.
 -
60
70  : Umschalten der Zeichengröße von 8 auf 16 Pixelzeilen.
80  : Beschränkung: 17 Zeilen / Bildschirm
90  : Darstellung von 16 an Stelle 8 Pixelzeilen / Zeichen
100 : Statt 40 nur noch 20 Zeilen (+Rahmen) für den Raster-
      strahl
110 : Bildschirminhalt ins richtige Format bringen
```

1.23. Doppelt hohe Zeichen

Was kann man nun mit dieser 16*8-Punktematrix anfangen? Als erste Anregung wollen wir die Größe der Zeichen auf dem Bildschirm verdoppeln. Dies erledigt das folgende BASIC-Programm. Voraussetzung ist allerdings, daß durch obiges Programm bereits auf 16*8-Punktematrix umgeschaltet worden ist.

(Programm 1.23.a)

Zugegeben: Die Sache ist in BASIC ziemlich zeitraubend. Deshalb eine ähnliche Routine in der wesentlich schnelleren Maschinensprache:

```
0B00 A2 03     LDX #$03       : Einleseschleife
0B02 BD 41 0B  LDA 0B41,X     : Anfangsadressen laden
0B05 95 FA     STA $FA,X      : in freien Zeropage-Raum
0B07 DEX                      : alles gelesen?
0B08 10 F8     BPL $0B02      : nein, weiter
0B0A A2 01     LDX #$01       : Bank-Konfig.
0B0C 8E 00 FF  STX $FF00      : setzen
0B0F A0 00     LDY #$00       : ab 0+Vektor
0B11 B1 FA     LDA ($FA),Y    : Anfangsadressen einlesen
0B13 48        PHA            : retten
0B14 A2 00     LDX #$00       : Bank-Konfig.
0B16 8E 00 FF  STX $FF00      : setzen
0B19 A6 FC     LDX $FC        : Lo Video-RAM Adresse
0B1B A4 FD     LDY $FD        : Hi Video-RAM Adresse
0B1D 20 46 0B  JSR $0B46      : POKE-Subroutine
0B20 E6 FC     INC $FC        : Lo=Lo+1
0B22 D0 02     BNE $0B26      : Lo noch größer 0
0B24 E6 FD     INC $FD        : Lo=0:Hi=Hi+1
0B26 68        PLA            : Hi ROM holen
0B27 A6 FC     LDX $FC        : Lo Video-RAM Adresse
0B29 A4 FD     LDY $FD        : Hi Video-RAM Adresse
0B2B 20 46 0B  JSR $0B46      : POKE-Subroutine
0B2E E6 FC     INC $FC        : Lo=Lo+1
0B30 D0 02     BNE $0B34      : Lo noch größer als 0
0B32 E6 FD     INC $FD        : Lo=0: Hi=Hi+1
0B34 E6 FA     INC $FA        : Lo ROM=Lo ROM+1
0B36 D0 02     BNE $0B3A      : noch größer 0?
0B38 E6 FB     INC $FB        : nein, Hi ROM=Hi ROM+1
0B3A A4 FB     LDY $FB        : laden
0B3C C0 E0     CPY #$E0       : Ende im ROM erreicht?
0B3E 90 CA     BCC $0B0A      : nein, weiter
0B40 60        RTS            : zurück zu BASIC
0B41 00 D0 00 20 00           : Zeiger Anfangsadressen
0B46 48        PHA            : Zeichen retten
0B47 8A        TXA
0B48 48        PHA            : Lo-Byte retten
0B49 98        TYA
0B4A 48        PHA            : Hi-Byte retten
0B4B A9 02     LDA #$02
0B4D 8D 28 0A  STA $0A28      : Cursor-Flag setzen
```

```
0B50 A2 12    LDX #$12      : REG 18 VDC
0B52 68       PLA           : Lo-Byte
0B53 20 61 0B JSR $0B61     : Register setzen
0B56 E8       INX           : REG 19 VDC
0B57 68       PLA           : Hi-Byte
0B58 20 61 0B JSR $0B61     : Register setzen
0B5B A2 1F    LDX #$1F      : REG 31 VDC
0B5D 68       PLA           : Zeichen
0B5E 4C 61 0B JMP $0B61     : Register setzen, fertig
0B61 8E 00 D6 STX $D600     : gewünschtes REG übergeben
0B64 2C 00 D6 BIT $D600     : Bit 7 gesetzt?
0B67 10 FB    BPL $0B64     : nein, also warten
0B69 8D 01 D6 STA $0601     : Wert übergeben
0B6C 60       RTS           : Zurück
```

Und gleich noch der dazugehörige BASIC-Loader:

(Programm 1.23.b)

Definition der 16*8-Matrix

Als weitere Anwendung wollen wir jetzt echte 16*8-Punkte-Zeichen definieren. Diesen Vorgang kennen Sie sicherlich aus dem Kapitel "Definition des eigenen Zeichensatzes" von der normalen 8*8-Matrix her. Wir werden für die 16*8-Matrix denselben Zeichen-Editor verwenden! Nehmen Sie hierzu lediglich folgende Änderung vor:

```
   2 MA=15

5055 FOR Y=0 TO 15
```

Viel Spaß beim Editieren der größeren Zeichen! Wollen Sie den dabei entstehenden BASIC-Loader in eigenen Programmen verwenden, beachten Sie bitte, daß zuvor auf 16*8-Matrix umgeschaltet werden muß. Fügen Sie deshalb einfach das vorletzte Programm, die Umschaltroutine, zum Loader hinzu.

1.24. Interne Verschiebungen im Video-RAM

Sehen Sie sich doch noch einmal die interne Aufteilung des Video-RAMs an: Drei Bereiche teilen sich den 16k großen Speicher:

```
2k Bildschirmspeicher
2k Attribut-RAM
8k Zeichengenerator
```

Was mit den restlichen freien 4k anzufangen ist, finden Sie an anderer Stelle in diesem Buch. Uns geht es hier vielmehr um die Verschiebung der einzelnen Bereiche. Nach dem Einschalten bietet sich Ihnen folgende Aufteilung:

```
$0000 Bildschirmspeicher
$0800 Attribut-RAM
$1000 frei
$2000 Zeichengenerator
```

Diese Aufteilung läßt sich jedoch von Ihnen beliebig verändern. Dabei ist es möglich, Bildschirm- und Attribut-RAM in 256-Byte-Schritten zu verschieben, während der Zeichengenerator nur in 8k-Blöcken verlegt werden kann. Zuständig hierfür ist wieder der VDC:

```
12  :  HI-Byte Bildschirmspeicher
13  :  LO-Byte Bildschirmspeicher

20  :  HI-Byte Attribut-RAM
21  :  LO-Byte Attribut-RAM

28  :  HI-Byte Zeichengenerator (Bits 5-7)
```

Sie sehen also: Anders als beim 40-Zeichen-Schirm kann hier auch der Farbspeicher verschoben werden!

Verschieben des Attribut-RAMs

Das folgende kleine Programm verschiebt das Attribut-RAM an eine beliebige Adresse im Video-RAM. Bitte beachten Sie, daß diese Verschiebung nur in 256-Byte Schritten erfolgen kann!
Auch sollten Sie das Attribut-RAM nur in freie Bereiche des Video-RAM verlegen, da ansonsten merkwürdige Effekte auftreten könnten.

```
10 REM ATTRIBUT-RAM VERSCHIEBER
20 INPUT "NEUE ANFANGSADRESSE";AD$
25 AD=DEC(AD$)
30 IF AD/256=INT(AD/256)THEN 40: ELSE PRINT "256K-
   SCHRITTE!": GOTO 20
40 HI=INT(AD/256): LO=AD-(256*HI)
50 A=DEC("D600"): B=A+1
60 POKE A,20: POKE B,HI
70 POKE A,21: POKE B,LO
80 POKE 2607,HI
90 END
```

Erwähnenswert erscheint uns die Zeile 80. Sollen Bildschirm- oder Attribut-RAM verschoben werden, so genügt es nicht, nur die entsprechenden VDC-Register zu verändern. Gleichzeitig muß auch jeweils eine bestimmte Adresse in der erweiterten Zeropage mit dem HI-Byte der neuen Anfangsadresse gefüttert werden:

```
Adresse 2606 : HI-Byte des Bildschirm-RAMs
Adresse 2607 : HI-Byte des   Attribut-RAMs
```

Geben Sie probeweise einmal die Adresse "1000" ein. Das Attribut-RAM wird in den freien Bereich gelegt. Geben Sie jetzt verschiedenfarbige Zeichen auf den Bildschirm aus! Rufen Sie die Verschieberoutine erneut auf und geben Sie diesmal wieder die ursprüngliche Adresse "0800" ein. Die farbigen Zeichen entfärben sich wieder und nehmen die Farbe des Bildschirmes vor dem Umschalten an.

Sie können so also für eine Bildschirmseite verschiedene Farbspeicher präparieren und dann gegebenenfalls umschalten! Als freie Anfangsadressen für weitere Attribut-RAMs empfehlen wir die Adressen "1000" und "1800"!

Um zu zeigen, was passiert, wenn das Attribut-RAM in einen nicht freien Bereich verlegt wird, geben Sie bitte als Anfangsadresse "0000" an; damit befindet sich das Attribut-RAM in demselben Adreßbereich wie das Bildschirm-RAM. Die Folge: Jedes Zeichen auf dem Bildschirm besitzt seine eigene Darstellungsfarbe und -Art!

Sofern Sie das Attribut-RAM in den Zeichengenerator legen (Adresse "2000"), verlieren entsprechende Zeichen ihr charakteristisches Äußeres, sobald Sie die Farbe verändern.

Verschieben des Bildschirm-RAMs

Dies funktioniert ganz analog zum Attribut-RAM. Die hierbei aktuellen Adressen entnehmen Sie bitte dem vorangegangenen Abschnitt.

Das Verschieben des BS-RAMs kann allerdings sinnvoller genutzt werden, wie wir meinen. Es lassen sich in Anbetracht des freien Speicherplatzes bis zu drei Bildschirmseiten programmieren, die unabhängig voneinander beschrieben werden können! Eine mögliche Anwendung entnehmen Sie bitte dem Kapitel "Arbeiten mit mehreren Bildschirmen".

1.25. Farbe für den 80-Zeichen-Schirm

Farbe? Na klar! 16 Farben stehen für Bildschirmzeichen zum Einfärben bereit. Sie lassen sich zum Beispiel durch CTRL 1 - 8 sowie C= 1 - 8 einschalten.

Wie wollen aber die Farbe des Bildschirmhintergrundes verändern. Auch dafür stehen 16 Farben zur Verfügung. Was beim 40-Zeichen-Schirm die Adressen 53280 und 53281 sind, ist beim 80-Zeichen-Schirm das Register 26:

26 Hintergrundfarbe

Mit diesem Register läßt sich der Bildschirmhintergrund leicht Ihren Wünschen entsprechend verfärben:

POKE DEC("D600"),26: POKE DEC("D601"),X (X = Farbe)

So, der Hintergrund ist bunt. Jetzt kommen wir zu den Zeichen auf dem Schirm.

Die Methode mit den Farbsteuerzeichen in PRINT-Anweisungen funktioniert zwar, ist aber nicht gerade die beste und übersichtlichste. Wir wollen Ihnen vorschlagen, künftig die Zeichenfarbe mit der Adresse 241 festzulegen (beim C64 Adresse 646). Das funktioniert so:

POKE 241,X (X = Farbnummer von 0 - 15)

Damit sind die Fähigkeiten dieser Adresse aber bei weitem noch nicht erschöpft. Sehen Sie sich doch einmal die letzten 4 Bits eines Bytes im Attribut-RAM an:

```
Bit 4 : blinken
Bit 5 : unterstreichen
Bit 6 : revers
Bit 7 : 2. Zeichensatz
```

Bisher war es schwierig, diese weiteren Funktionen des Attribut-RAMs bei der eigenen Programmierung voll zu nutzen.

Die Umstellung von einem auf den anderen Zeichensatz funktionierte zur Not auch durch gleichzeitiges Drücken der Tasten C= + SHIFT. Auch die reverse Ausgabe einer PRINT-Anweisung war bisher mit einem Steuerzeichen recht leicht zu regeln. Gerade die völlig neuen Funktionen des Attribut-RAMs, "Blinken" und "Unterstreichen" blieben dabei leider auf der Strecke.

Sie sollten diese nützlichen Funktionen jedoch bei der eigenen Programmierung voll nutzen! Es lohnt sich wirklich. Gehen Sie dabei einfach so vor:

POKE 241,PEEK(241) OR 2^4: PRINT"DIESE ZEILE BLINKT!"

sorgt dafür, daß die nachfolgende PRINT-Anweisung blinkend ausgegeben wird.

POKE 241,PEEK(241) OR 2^5: PRINT "UNTERSTRICHEN!"

unterstreicht automatisch die nachfolgende Zeile. Eine mögliche Anwendung wäre:

```
10 PRINT "Diese Tatsache ist";
20 POKE 241,PEEK(241)OR 2^5
30 PRINT "wichtig";
40 POKE 241,PEEK(241) AND NOT 2^5
50 PRINT "!"
60 END
```

Wenn Sie "2^5" durch "2^4" ersetzen, blinkt das wichtige Wort. Ersetzen Sie "2^5" durch "2^5+2^4", dann wird das Wort gleichzeitig unterstrichen und blinkt.

Sie können natürlich auch "2^6" (reverse) und / oder "2^7" (2.Zeichensatz) benutzen!

Mit dieser Programmiertechnik sollte es Ihnen nicht schwerfallen, wirklich ansprechende Software zu schreiben!

1.26. Ein zusätzlicher Zeichengenerator

Eingangs dieses Kapitels erklärten wir den Aufbau des Zeichengenerators im Video-RAM des 80-Zeichen-Controllers. Hauptmerkmal war, daß für jedes Zeichen statt wie gewöhnlich 8 hier 16 Bytes reserviert sind.

Diese 16 Bytes wurden sinnvoll bereits in Verbindung mit der 16*8-Punkte-Definition verwendet. Wir wollen Ihnen hier einen weiteren Vorschlag machen, was damit anzufangen ist.

Voraussetzung ist, daß Sie mit der normalen 8*8-Punktematrix pro Zeichen auskommen. Pro Zeichen werden dann nur 8 Bytes benutzt, die anderen 8 Bytes stehen unbenutzt in der Gegend herum. Aber nicht mehr lange. Die folgende Maschinenroutine "Swapper" vertauscht die ersten acht mit den folgenden acht Bytes innerhalb der 16 Bytes pro Zeichen. Es kann also ein zusätzlicher Zeichengenerator definiert werden, der sich in den unbenutzten acht Bytes pro Zeichen herumlümmelt und auf Verlangen in den aktiven Bereich umkopiert wird!

Zunächst das Programm:

```
0B00 A9 00     LDA #$00      : Lo-Byte erster Zeiger
0B02 85 4C     STA $4C       : abspeichern
0B04 A9 20     LDA #$20      : Hi-Byte erster Zeiger
0B06 85 4D     STA $4D       : abspeichern
0B08 A9 08     LDA #$08      : Lo-Byte zweiter Zeiger
0B0A 85 4E     STA #$4E      : abspeichern
0B0C A9 20     LDA #$20      : Hi-Byte zweiter Zeiger
0B0E 85 4F     STA $4F       : abspeichern
0B10 A0 07     LDY #$07      : 8 Bytes
0B12 98        TYA
0B13 48        PHA           : auf Stack
0B14 A5 4C     LDA $4C       : Lo-Byte erster Zeiger
0B16 A6 40     LDX $4D       : Hi-Byte erster Zeiger
0B18 20 8C 0B  JSR $0B8C     : PEEK-Subroutine
0B1B 48        PHA           : gelesenes Zeichen auf Stack
0B1C A5 4E     LDA $4E       : Lo-Byte zweiter Zeiger
0B1E A6 4F     LDX $4F       : Hi-Byte zweiter Zeiger
```

```
0B20 20 8C 0B JSR $0B8C     : PEEK-Subroutine
0B23 A6 4C    LDX $4C       : Lo-Byte erster Zeiger
0B25 A4 4D    LDY $4D       : Hi-Byte erster Zeiger
0B27 20 65 0B JSR $0B65     : POKE-Subroutine
0B2A 68       PLA           : gelesenes Zeichen
0B2B A6 4E    LDX $4E       : Lo-Byte zweiter Zeiger
0B2D A4 4F    LDY $4F       : Hi-Byte zweiter Zeiger
0B2F 20 65 0B JSR $0B65     : POKE-Subroutine
0B32 E6 4C    INC $4C       : Lo-Byte1=Lo-Byte1+1
0B34 D0 02    BNE $0B38     : noch größer als 0?
0B36 E6 4D    INC $4D       : nein, Hi1=Hi1+1
0B38 E6 4E    INC $4E       : Lo-Byte2=Lo-Byte2+1
0B3A D0 02    BNE $0B3E     : noch größer als 0?
0B3C E6 4F    INC $4F       : nein, Hi2=Hi2+1
0B3E 68       PLA           : gelesener Wert zurück
0B3F A8       TAY
0B40 88       DEY
0B41 10 CF    BPL $0B12     : weiter kopieren
0B43 A5 4F    LDA $4F       : Hi-Byte zweiter Zeiger
0B45 C9 40    CMP #$40      : $4000 erreicht?
0B47 B0 1B    BCS $0B64     : ja, fertig
0B49 A9 08    LDA #$08      : Lo=Lo+8 Zeichenbytes
0B4B 65 4C    ADC $4C       : addieren
0B4D 85 4C    STA $4C       : wieder abspeichern
0B4F A9 00    LDA #$00      : 0 addieren
0B51 65 4D    ADC $4D       : Carry-Übertrag addieren
0B53 85 4D    STA $4D       : wieder speichern
0B55 A9 08    LDA #$08      : Lo2=Lo2+8 Zeichenbytes
0B57 65 4E    ADC $4E       : addieren
0B59 85 4E    STA $4E       : wieder speichern
0B5B A9 00    LDA #$00      : 0 addieren
0B5D 65 4F    ADC $4F       : Carry-Übertrag addieren
0B5F 85 4F    STA $4F       : wieder abspeichern
0B61 4C 10 0B JMP $0B10     : Schleife
0B64 60       RTS           : Zurück zu BASIC
0B65 48       PHA           : Zeichen retten
0B66 8A       TXA
0B67 48       PHA           : Lo-Byte retten
0B68 98       TYA
0B69 48       PHA           : Hi-Byte retten
```

```
0B6A A9 02     LDA #$02
0B6C 8D 28 0A  STA $0A28     : Cursorflag setzen
0B6F A2 12     LDX #$12      : REG 18 VDC
0B71 68        PLA           : Hi-Byte zurückholen
0B72 20 80 0B  JSR $0B80     : Register setzen
0B75 E8        INX           : REG 19 VDC
0B76 68        PLA           : Lo-Byte zurückholen
0B77 20 80 0B  JSR $0B80     : Register setzen
0B7A A2 1F     LDX #$1F      : REG 31 VDC
0B7C 68        PLA           : Byte holen
0B7D 4C 80 0B  JMP $0B80     : fertig, Register setzen
0B80 8E 00 D6  STX $0600     : REG setzen
0B83 2C 00 D6  BIT $D600     : warten
0B86 10 FB     BPL $0B83     : genug gewartet?
0B88 8D 01 D6  STA $D601     : Wert übergeben
0B8B 60        RTS           : fertig
0B8C 48        PHA           : Lo-Byte retten
0B8D 8A        TXA
0B8E 48        PHA           : Hi-Byte retten
0B8F A2 12     LDX #$12      : REG 18 VDC
0B91 68        PLA           : Hi-Byte in Akku
0B92 20 80 0B  JSR $0B80     : Register setzen
0B95 E8        INX           : REG 19 VDC
0B96 68        PLA           : Lo-Byte in Akku
0B97 20 80 0B  JSR $0B80     : Register setzen
0B9A A2 1F     LDX #$1F      : REG 31 VDC
0B9C 8E 00 D6  STX $D600     : Register setzen
0B9F 2C 00 D6  BIT $D600     : warten
0BA2 10 FB     BPL 0B9F      : genug gewartet?
0BA4 AD 01 D6  LDA $D601     : Wert aus Video-RAM lesen
0BA7 85 FE     STA $FE       : ablegen
0BA9 60        RTS           : fertig
```

Und der zugehörige BASIC-Loader:

(Programm 1.26.a)

Nach dem Aufruf der Routine durch SYS DEC("0B00") färbt sich der Bildschirm schwarz. Schließlich meldet sich der Cursor wieder.

Der Zeichengenerator wurde ordnungsgemäß umgeschaltet. Die zweiten acht Bytes eines Zeichens enthalten aber normalerweise Null. Also sind nach dem Umschalten alle Zeichen als Leerzeichen definiert. Schalten Sie wieder auf den ursprünglichen Zeichensatz durch erneuten Aufruf der Routine. Laden Sie jetzt den Zeichen-Editor des Kapitels "Definition des eigenen Zeichensatzes". Sie können jetzt nach Herzenslust Ihren eigenen Zeichensatz entwickeln. Im dabei entstehenden BASIC-Loader verändern Sie aber bitte Zeile 5065 in:

```
5065 AD=8192+16*A+Y+8
```

Starten Sie nun den Loader. Offensichtlich haben alle Zeichen ihr Äußeres behalten. Swappen Sie nun mit unserer Routine den Zeichengenerator. Nun steht Ihnen der neudefinierte Zeichensatz zur Verfügung. Gleichzeitig können Sie aber jederzeit auf den alten Zeichensatz zurückgreifen!

Denkbar sind natürlich auch zwei selbstdefinierte Sätze, die den Umständen entsprechend ein- und ausgeschaltet werden können (z. b. für Spiele etc.).

1.27. Systemroutinen für den Hausgebrauch

Wollen Sie eigene Anwendungen mit dem 80-Zeichen-Bildschirm schreiben, so werden Sie oft wünschen, schon fertige Routinen zu haben, mit denen Sie bestimmte Aktionen durchführen können - sei es ein Zugriff auf das Video-RAM oder ein Neu-Initialisieren des Controllers.
Da dafür schon einige fertige Routinen im Betriebssystem existieren, können Sie sich viel Zeit, Mühe und Speicherplatz sparen.

Aber auch für die Benutzer des 40-Zeichen-Bildschirms können die hier vorgestellten ROM-Routinen durchaus wertvolle Dienste leisten, da sie nicht speziell für eine der beiden Darstellungsarten geschrieben wurden (das Betriebssystem muß ja auch auf beide Bildschirme zurückgreifen können), sondern intern unterschieden wird, welcher Bildschirm angesprochen werden soll. Durch die Anwendung der folgenden ROM-Routinen ist es also möglich, Programme zu schreiben, die vom gerade benutzten Bildschirm weitgehend unabhängig sind und so ein viel breiteres Anwenderspektrum erreichen.

Doch nun zu den eigentlichen Routinen. Sie finden jeweils zwei Einsprungadressen. Die erste bezieht sich auf die Jump-Tabelle für den Editor. Die zweite Adresse ist die Anfangsadresse der eigentlichen Routine. Welche Adresse Sie anspringen wollen, bleibt natürlich Ihnen überlassen.

Initialisierung des Bildschirms

Die folgende Routine initialisiert den Bildschirm. Er wird also wieder in den Normalzustand versetzt. Alle zuvor vorgenommenen Manipulationen werden damit wieder nichtig. Es erscheint der Bildschirm so, wie er vom Einschalten her bekannt ist. Dabei ist es egal, welcher der beiden Bildschirme gerade benutzt wurde:

```
SYS 49152 / SYS 49275
```

Ausgabe eines Zeichens mit Farbe

Diese Routine wurde bereits im Kapitel "Zuverlässiger Video-RAM-Zugriff" in Zusammenhang mit dem 80-Zeichen-Bildschirm näher beschrieben. Die Routine setzt ein Zeichen mit Farbe auf den Bildschirm. Die Position ist abhängig vom Inhalt der Speicherstellen 224 und 225 der Zeropage:

```
SYS 49155,Zeichen,Farbe
SYS 52276,Zeichen,Farbe

Zeichen: (0-255) Bestimmt das auszugebende Zeichen im
                 Bildschirmcode (A=1, B=2, C=3, etc.
Farbe:   (0-15)  Legt die Farbe des auszugebenden Zeichens
                 fest (0=schwarz, 1=weiß, 2=rot etc.)
                 Im 80-Zeichen-Mode sind hier auch Werte
                 von 0 bis 255 zulässig. Die zusätzlichen
                 vier Bits haben dann folgende Bedeutung:

                 Bit 4: Blinken
                 Bit 5: Unterstreichen
                 Bit 6: Reverse
                 Bit 7: 2. Zeichensatz

                 Ein Wert von 2^0+2^4=1+16=17 ergäbe (im 80-
                 Zeichen-Mode) ein weißes,  blinkendes
                 Zeichen.
```

Ausgabe eines ASCII-Zeichens

Die vorangegangene Routine gab ein Zeichen aus, das im sogenannten "Bildschirmcode" festgelegt war. Dieser Code ist Commodore-spezifisch und kein Standard. Diese Routine gibt ein Zeichen im ASCII-Code aus:

```
SYS 49164,ASCII-Code
SYS 50989,ASCII-Code
```

Die Routine entspricht also in gewisser Weise

```
PRINT CHR$(ASCII-Code)
```

Der ASCII-Code eines Zeichens läßt sich zum Beispiel so ermitteln:

```
PRINT ASC("K")
```

gibt den ASCII-Code des Zeichens "K" aus.

PRINT AT simuliert

Sicherlich werden viele Maschinensprache-Programmierer gerne in den Genuß des BASIC-Befehls CHAR kommen. Mit ihm kann man ein Zeichen an beliebiger Stelle auf dem Bildschirm darstellen lassen (der Befehl heißt auch PRINT AT bei anderen BASIC-Versionen). Glücklicherweise existiert auch hierfür eine ROM-Routine (muß ja, denn wie würde CHAR das Zeichen sonst auf den Bildschirm bringen?):

```
SYS 49176,A,Spalte,Zeile:PRINT...
SYS 52330,A,Spalte,Zeile:PRINT...
```

Mit dieser Routine könnten Sie beispielsweise eine Kommando-Zeile errichten, d.h., in einer bestimmten Bildschirmzeile wird immer die aktuelle Meldung an den Anwender ausgegeben etc. So etwas findet man sehr häufig in professioneller Software; es erhöht die Übersicht und Anwenderfreundlichkeit erheblich!
Wollen Sie nach PRINT AT wieder zu der Stelle zurückspringen, von wo Sie gekommen sind, dann müssen Sie lediglich die aktuelle Cursorposition vor PRINT AT zwischenspeichern. Dies könnte so aussehen:

```
10 REM PRINT AT mit Rücksprung
20 SP=PEEK(236): REM aktuelle Spalte speichern
30 ZE=PEEK(235): REM aktuelle Zeile speichern
```

```
40 :
50 SYS 49176,0,5,10: PRINT"Kommandozeile"
60 :
70 SYS 49176,0,SP,ZE: REM wieder zurück
80 PRINT"Wieder da"
90 END
```

Diese Routine wird übrigens vom Betriebssystem benutzt, um die aktuelle Cursorposition zu holen. Welche Funktion sie erfüllt, hängt vom Zustand des Carry-Flags ab (Carryflag gelöscht = Cursorposition setzen / Carryflag gesetzt = Cursorposition setzen).

Neudefinition des Zeichensatzes

Diese Routine ist ausnahmweise einmal nur für den 80-Zeichen-Bildschirm zuständig.
Falls Sie den hauseigenen Zeichensatz durch eigene Kreationen verändert haben (siehe "Definition des eigenen Zeichensatzes"), Ihnen Ihr neuer Zeichensatz aber nicht so recht gefällt oder Sie aus irgendeinem anderen Grund lieber wieder mit dem ursprünglichen Zeichensatz arbeiten möchten, so spielt diese Routine Retter in der Not, da sie den Original-Zeichensatz neu in das Video-RAM des VDC kopiert.

```
SYS 49191 / SYS 552748
```

40/80-Zeichen-Umschaltung

Diese Routine wurde zu Anfang dieses Kapitels schon einmal erwähnt. Sie schaltet vom 40- in den 80-Zeichen-Bildschirm und umgekehrt:

```
SYS 49194 / SYS 52526
```

Nähere Erklärungen zu dieser Routine finden Sie im Kapitel 11.2.: "Kernal".

1.28. Hochauflösende Grafik

Auch zum Thema "hochauflösende Grafik" hat das neue und leistungsfähige V7.0-BASIC des C128 eine Menge zu bieten. Uns erstaunte jedoch sehr, daß sämtliche Grafik-Befehle ausschließlich auf dem 40-Zeichen-Bildschirm wirksam sind.
Weshalb der 80-Zeichen-Grafikbildschirm mit seiner doppelt großen Auflösung von 640*200 Punkten gegenüber 320*200 Punkten des 40-Zeichen Bildschirmes vom neuen Basic nicht berücksichtigt wird, können wir nicht verstehen. Wir glauben aber, daß es für Sie wichtig ist, auch diesen Grafikbildschirm programmieren zu können, wo er einmal vorhanden ist!

Auf den folgenden Seiten wollen wir Ihnen zeigen, wie dies zu bewerkstelligen ist und welche Anwendungen sich daraus ergeben. Arbeiten Sie ohnehin ausschließlich mit einem RGB-Monitor, so kommen Sie nun auch in den Genuß hochauflösender Grafik (es wäre wirklich ein "Rückschritt", wenn RGB-Monitor-Benutzer zusätzlich einen Fernsehapparat an den Rechner anschließen müßten, nur weil keine Grafik-Befehle für den 80-Zeichen-Schirm existieren!).

Der Bit-Map-Mode

Wer auf dem Commodore 64 schon einmal in hochauflösender Grafik programmiert hat, wird den sogenannten "Bit-Map-Mode" kennen: Er schaltet im Prinzip von normaler Bildschirmdarstellung auf hochauflösende Grafik um. Das Video-RAM ist dann nicht mehr unterteilt in Bildwiederholspeicher, Attribut-RAM und Zeichengenerator. Vielmehr klappert der Rechner nun Bit für Bit des Video-RAMs ab. Für jedes gesetzte Bit wird so ein Punkt auf den Schirm geschrieben.
Für die Auflösung des 80-Zeichen-Bildschirms würde das heißen:

16000 Bytes * 8 Bits = 128.000 Bildpunkte

welche einzeln gesetzt oder gelöscht werden können!

Wir wollen zunächst einmal den Bit-Map-Mode einschalten. Dazu werden die Register 20 und 25 benötigt:

```
20  Attribut-RAM Startadresse HI
25  Bit 7: HIRES ein/aus
```

Die folgenden Zeilen schalten den Rechner in den Bit-Map-Mode:

```
10 A=DEC("D600"):B=A+1
20 POKE A,20:POKE B,0
30 POKE A,25:POKE B,128
```

Sofort erscheint auf dem Bildschirm ein wüstes Durcheinander an Punkten und Strichen. Dieses Chaos hat aber System. Jedes gesetzte Bit in den 16k Video-RAM stellt nun einen Punkt dar!

Was ist aber mit diesem Grafikbildschirm anzufangen? Alle Grafik-Befehle wirken auf den 40-Zeichen-Schirm.

Selbst ist der Programmierer! Benutzen Sie doch einfach die modifizierte POKE-Anweisung der vorangegangenen Seiten!
Geben Sie ein:

```
SYS DEC("1800"),128,0,0
```

Damit schreiben Sie ein kleines Pünktchen in die linke obere Ecke des 80-Zeichen-Grafikschirms.
Dies aber nur des Prinzips wegen. Das folgende Programm zeichnet eine Sinuskurve auf den Schirm.

Die PLOT-Routine arbeitet der höheren Geschwindigkeit wegen mit einigen Maschinenroutinen. Es handelt sich dabei um:

a) modifiziertes POKE (siehe auch "POKE-Simulation")
b) modifiziertes PEEKc) ERASE (Löschen des Grafikschirms)

Tippen Sie also zunächst das folgende Programm ab und starten Sie es. Es handelt sich dabei um die drei entsprechenden BASIC-Loader, die die benötigten Routinen initialisieren!

(Programm 1.28.a)

Und hier die Sinuskurve:

(Programm 1.28.b)

Die modifizierte POKE-Routine wurde bereits an anderer Stelle dokumentiert. Hier also die ERASE-Routine:

ERASE - Löschen des 80-Zeichen-Grafik-Schirms

```
1900 A2 40    LDX #$40      : 64 Pages löschen
1902 A9 00    LDA #$00      : ab $0000
1904 A0 00    LDY #$00
1906 85 FE    STA $FE       : Hi-Byte abspeichern
1908 48       PHA           : und retten
1909 8A       TXA
190A 48       PHA           : Pages retten
190B A2 12    LDX #$12      : REG 18 VDC
190D A5 FE    LDA $FE       : Hi-Byte laden
190F 20 2A 19 JSR $192A     : Register setzen
1912 E8       INX           : REG 19 VDC
1913 98       TYA           : Lo-Byte in Akku
1914 20 2A 19 JSR $192A     : Register setzen
1917 A2 1F    LDX #$1F      : REG 31 VDC
1919 A9 00    LDA #$00      : Null ins Video-RAM schreiben
191B 20 2A 19 JSR $192A     : Register setzen
191E 68       PLA           : Hi-Byte holen
```

```
191F AA       TAX          : ins X-Reg.
1920 68       PLA          : Pages holen
1921 C8       INY
1922 D0 E4    BNE $1908    : Schleife
1924 E6 FE    INC $FE      : Hi=Hi+1
1926 CA       DEX          : Page=Page-1
1927 D0 DF    BNE $1908    : noch nicht 0? Dann weiter!
1929 60       RTS          : zurück zu BASIC
192A 8E 00 D6 STX $D600    : Register setzen
192D 2C 00 D6 BIT $D600    : warten
1930 10 FB    BPL $192D    : genug gewartet?
1932 80 01 D6 STA $D601    : Wert übergeben
1935 60       RTS          : zurück
```

Diese Grafik-Befehle stellen natürlich nicht das "Non-plus-ultra" dar; dafür sind sie zu langsam und zu wenig. Sie sollten jedoch auch nur als Anregung dienen, so daß sich wahrscheinlich innerhalb kurzer Zeit zahlreiche Grafik-Befehls-Erweiterungen auf dem Markt befinden werden.

Schreiben Sie doch selbst einmal eine!

1.29. Zeichengenerator einmal anders

Zum Schluß noch ein kleines Programm, daß Sie garantiert "kopfstehen" läßt - im wahrsten Sinne des Wortes.
Doch probieren Sie es einfach aus. Voraussetzung ist allerdings, daß sich die POKE-Routine (ab $1800) und die PEEK-Routine (ab $1A00) aus den vorhergehenden Kapiteln im Speicher befinden.

(Programm 1.29.a)

Nun das ganze auch noch für die 16*8-Matrix:

(Programm 1.29.b)

2. GRAFIK MIT DEN EINGEBAUTEN BEFEHLEN

2. Grafik mit den eingebauten Befehlen

****

2.1. Der CIRCLE-Befehl

Der CIRCLE-Befehl ist wohl einer der vielseitigsten Befehle, die das BASIC V7.0 enthält. Der Name "CIRCLE" gibt nicht an, was dieser Befehl wirklich leisten kann: außer Kreisen und Geraden auch verschiedenartige Dreiecke, Vierecke und Ellipsen.

Der CIRCLE-Befehl hat folgendes Format:

```
CIRCLE fs,x,y,xr,yr,sw,ew,w,i
```

Die Parameter hinter dem Befehl haben folgenden Bedeutung:

fs	Nummer des Farbspeichers (0 - 3)
x,y	Koordinaten des Kreismittelpunktes
xr	Länge des Radius in X-Richtung
yr	Länge des Radius in Y-Richtung
sw	Anfangswinkel des Kreises
ew	Endwinkel des Kreises
w	Winkel für Rotation
i	Winkel für die zu zeichnenden Kreissegmente

Um einen Kreis zu zeichnen, brauchen nur die ersten vier Parameter eingegeben zu werden; der Rest kann entfallen. Der erste Wert gibt den Farbspeicher, die nächsten beiden die Mittelpunktskoordinaten und der vierte Wert den Radius an.
Um eine Ellipse zu zeichnen, muß der Radius in X-Richtung ungleich dem Radius in Y-Richtung sein.
Beispiel:

```
CIRCLE 0,160,100,10,30
```

Wollen Sie die Ellipse um den Mittelpunkt drehen, müssen Sie den vorletzten Wert verändern. Beispiel:

CIRCLE 0,160,100,10,30,0,360,45

Dieser Parameter gibt die Gradzahl an, um die der Kreis (oder wie in diesem Fall die Ellipse) gedreht werden soll. Der sechste und der siebte Wert geben an, bei welcher Gradzahl der Kreis anfängt bzw. endet. Damit ließe sich beispielsweise ein Halbkreis zeichnen:

CIRCLE 0,160,100,30,30,0,180

Wie jedoch könnte ein Viereck gezeichnet werden? Dafür ist der letzte Parameter gedacht. Durch ihn können zahlreiche verschiedene Figuren dargestellt werden.
Die Bedeutung einzelner Werte:

0-44	Kreis (je größer der Wert desto "runder")
45	8-Eck
60	6-Eck
75	5-Eck
90	Gleichseitiges Viereck (Rauten, Quadrate)
91-119	Unregelmäßige Vierecke
120	Gleichseitiges Dreieck
121-179	Dreieck (je größer der Wert desto spitzwinkliger das Dreieck)
180-255	Geraden

2.2. Tortengrafik

Die Grafik-Befehle des BASIC V7.0 fordern natürlich Anwendungen geradezu heraus. Durch die vielfältigen Möglichkeiten des CIRCLE-Befehls läßt sich auf einfache Weise ein faszinierendes Programm schreiben: Tortengrafik.
Wer kennt sie nicht, die zahlreichen Werbungen für Kalkulationsprogramme? Die grafischen Möglichkeiten werden meistens durch ein Bild belegt; und meistens zeigt dieses Bild eine Tortengrafik. Nun sollen auch Sie als C-128er-Besitzer in den Besitz solch einer Tortengrafik kommen. Natürlich ist unser Programm nicht so ausgefeilt wie die fertigen Programme, die Si kaufen können. Verbessern Sie es doch!

Nun zu dem Programm: Dank des CIRCLE-Befehls artet eine Tortengrafik nicht in eine Sammlung wilder Formeln mit zahlreichen Sinus- und Cosinusberechnungen aus.
Ein Problem stellte sich nur dadurch, daß der Farbspeicher bei Hires-Grafik nicht die gleiche Größe wie die Grafik an sich hat. Dadurch wird immer gleich ein ganzes Feld mit 8 * 8 Punkten eingefärbt. Dem kann zwar durch den Multicolor-Modus abgeholfen werden, jedoch entsteht dann durch die geringere Auflösung ein neues Problem: eine eckige Torte.
All diese Probleme werden umgangen, in dem nur jedes zweite Segment ausgefüllt wird. Man kann natürlich das Ausmalen der einzelnen Segmente auch ganz ausschalten. Löschen Sie dafür die Zeilen 340 - 390.

(Programm 2.2.a)

Die Eingabe der verschiedenen Anteile erfolgt in den absoluten Werten und nicht in Prozent. Zu jedem Anteil können Sie einen Text angeben. Dieser sollte nicht zu lang werden, da sonst die Übersicht verloren geht.

2.3. Die Balkengrafik

Im vorigen Kapitel wurde ein Programm vorgestellt, das eine Tortengrafik erstellt. Jetzt möchten wir Ihnen noch ein weiteres Kalkulationsprogramm vorstellen: eine Balkengrafik.

Bei diesem Balkengrafik-Programm sind nicht mehr als 8 Balken vorgesehen, da sonst die Übersichtlichkeit auf dem 40-Zeichen-Bildschirm verloren gehen würde.
Die Länge dieser Balken richtet sich natürlich nach den von Ihnen eingegebenen Werten.
Die Werte werden so umgerechnet, daß der größte Wert 100 Prozent entspricht und die anderen Werte entsprechend darunter bleiben. Dadurch wird die vertikale Auflösung immer optimal ausgenutzt. Diese Umrechnung geschieht in Zeile 280.

(Programm 2.3.a)

Zuerst werden Sie nach einer Überschrift gefragt. Diese erscheint dann in der Grafik als Kopfzeile.
Zusätzlich kann jedem Balken noch eine Farbe zugewiesen werden.

2.4. Funktionsplotter

Wer kennt sie nicht, die elendige Kurvendiskussion? Die langwierigen Berechnungen zu Maxima, Minima und Wendestellen? Die endlosen Tüfteleien zu den einzelnen Tangenten?
Hier soll nun kein komplettes Programm zur Kurvendiskussion vorgestellt werden. Dennoch stellt es eine große Hilfe da: Es wird der Graph einer von Ihnen eingegebenen Funktion auf dem Bildschirm darstellt. Dadurch können Sie schnell die von Ihnen ausgerechneten Werte auf ihre Richtigkeit überprüfen.
Nun das Programm:

(Programm 2.4.a)

Nach dem Starten des Programms werden von Ihnen einige Werte verlangt. Zuerst einmal müssen Sie den Bereich angeben, in dem Sie die Funktion sehen wollen. Danach werden dann die Einheiten festgelegt, so daß der Ausschnitt immer so groß wie möglich auf dem Bildschirm zu sehen ist.
Für die vorgegebene Funktion sind z. B. folgende Werte geeignet: in X-Richtung ein Intervall von -5 bis 9 und in Y-Richtung ein Intervall von 0 bis 9.

Wollen Sie eine andere Funktion untersuchen, so müssen Sie die Zeile 30 ändern. Um eine Sinuskurve darzustellen, müßten Sie z. B. folgendes eingeben:

```
30 DEF FN Y(x)= SIN(x)
```

Bei dieser Kurve sollten Sie darauf achten, daß der Computer

nicht in Grad, sondern in Bogenmaß rechnet (360 Grad sind genau 2 * Pi Bogenmaß). Als Bereich eignet sich folgender: kleinster X-Wert = 0 und größter X-Wert gleich 3.1415. In Y-Richtung muß, wenn der ganze Graph auf dem Bidschirm erscheinen soll, nur das Intervall von -1 bis 1 angegeben werden.

2.5. Windows

"Windows" ist seit neuestem ein Schlagwort in der Computer-Branche. Kein neuer Rechner, der sie nicht besitzt, keine BASIC-Version, die sie nicht unterstützt. Da kam natürlich auch Commodore nicht umhin, dem 128er die Fähigkeit zum Umgang mit Windows mitzugeben.
Leider steht auf dem C-128 nur ein Fenster zur Verfügung.
Die Möglichkeit von vornherein auf mehrere Windows zugreifen zu können, wäre durchaus sinnvoll gewesen.
Nichts desto trotz experimentierten wir mit dieser Errungenschaft und schon nach kurzer Zeit des Ausprobierens kamen uns zahlreiche Ideen, mit denen man die Fähigkeiten des WINDOW-Befehls ausschöpfen kann.

2.5.1 Wie man ein Fenster verläßt

Wenn Sie Ihre Zeit dem Handbuch gewidmet haben sollten, ist Ihnen sicherlich bekannt, daß man den Rahmen eines Windows sowohl vom Programm als auch aus dem Direktmodus heraus setzen kann. Dies geschieht ohne Probleme. Aber wie kommt man aus dem Fenster wieder heraus? Mit den bekannten Cursorfunktionen stößt man da an die Fenstergrenzen. Das einfachste, was man in diesem Fall machen kann ist die Run-Stop/Restore Taste zu drücken. Durch dieses altbewährte Heilmittel wird der Bildschirm neu initialisiert und das eingegebene Window verschwindet.
Diese Operation hat aber leider den relativ schwewiegenden Nachteil, ein etwaiges Programm zu unterbrechen.

Zum Glück haben wir aber noch eine weitere Lösung auf Lager:

Im Prinzip ist ein Window ja nur ein verkleinerter Bildschirm und der Bildschirm ein Window mit der größtmöglichen Ausdehnung. Also braucht man, wenn man ein Window verlassen möchte, nur das Fenster auf die Größe des ganzen Bildschirms zu setzen, wie es rein zufällig auch normalerweise der Fall ist, und hat auf diese Weise das beengende Window verlassen:

```
WINDOW 0,0,79,24   (für 80-Zeichen-Bildschirm)
WINDOW 0,0,39,24
(für 40-Zeichen-Bildschirm)
```

2.5.2 Abfragen der Window-Koordinaten

Für diesen Zweck haben die Programmierer des BASICs dem Computer schon einen Befehl mitgegeben. Dieser hört auf den wunderschönen Namen RWINDOW.
Doch können Sie mit diesem Befehl nur die Zeilen (durch RWINDOW (0)), die Spalten (durch RWINDOW (1)) und den Zeichenmodus (40 oder 80 Zeichen durch RWINDOW(2)) in Erfahrung bringen. Wollen Sie die genauen Koordinaten wissen, müssen Sie anders handeln.

In der Zeropage wurden extra vier Bytes eingerichtet, in denen die Grenzwerte des momentanen Windows gespeichert sind. Dies sind die Adressen 228 - 231:

```
10 WINDOW 1,11,20,22
20 PRINT "UNTERE GRENZE:";PEEK(228)
30 PRINT "OBERE  GRENZE:";PEEK(229)
40 PRINT "LINKE  GRENZE:";PEEK(230)
50 PRINT "RECHE  GRENZE:";PEEK(231)
```

Die jeweilige Funktion wird durch das Programm deutlich. Damit Sie nicht in die Versuchung kommen zu glauben, die Werte würden zufällig übereinstimmen, gehen Sie mit RUN-STOP/RESTORE wieder aus dem Window heraus und ändern Sie die Parameter des Window-Befehls. Wie Sie sehen, ist die Ausgabe immer noch entsprechend den geänderten WINDOW-Werten.

2.5.3. Alternatives Fenster setzen

Um jene freche Behauptung von vorhin (Kapitel: "Wie man ein Fenster verläßt"), der Bildschirm sei einfach nur ein WINDOW, zu belegen, sollten wir uns noch einmal die Werte der Adressen 228 ($E4) bis 231 ($E7) anschauen. Die folgenden Werte gelten für den Normalzustand, ohne ein gesetztes WINDOW:

```
228 ($E4) :     24
229 ($E5) :      0
230 ($E6) :      0
231 ($E7) :     79   (Bei 80-Zeichen Bildschirm)
                39   (Bei 40-Zeichen Bildschirm)
```

Geben wir nun den Befehl

WINDOW 1,2,3,14

ein, so ändern sich die Werte

```
228 ($E4)       14
229 ($E5)        2
230 ($E6)        1
231 ($E7)        3
```

(unabhängig von der Bildschirmgröße)

Durch diese Errungenschaft kann man gleich die Frage beantworten, wie denn der Assembler-Programmierer die Fenster-Programmierung nutzen kann. Nun, es müssen nur die gewünschten Werte in die Adressen 228 - 231 gespeichert werden:

POKE 228,10

setzt die untere Grenze auf 10.

Sie sollten jedoch beachten, daß es zu unangenehmen Nebenerscheinungen kommen kann: z. B. wenn Sie die Fenstergröße außerhalb der möglichen Größe setzen (etwa im 40-Zeichen-Bildschirm in die Adresse 231 einen Wert über 39 schreiben oder eine Zeilenanzahl über 24 angeben).

2.5.4. Vertikale Laufschrift

Mit dem Befehl WINDOW kann man ohne großen Aufwand eine vertikale Laufschrift erzeugen.
Das Programm dazu sieht wie folgt aus:

```
 0 REM Laufschrift
10 INPUT "TEXT";A$
20 INPUT "GESCHWINDIGKEIT";G
30 WINDOW 10,10,10,20
40 FOR I=1 TO LEN(A$)
50 PRINT MID$(A$,I,1)
60 FOR T=1 TO G
70 NEXT T
80 NEXT I
90 GOTO 40
```

Die Breite des Fensters wird auf eins reduziert. Dadurch werden alle Buchstaben untereinander ausgegeben - und es entsteht eine Laufschrift.
Leider funktioniert dieses System nicht bei einem waagerecht angelegten Fenster.

2.5.5. Das Fenster als Eingabebegrenzung

Mit dem WINDOW-Befehl läßt sich ganz einfach verhindern, daß ein Benutzer während einer Eingabe mittels der Cursortasten die Eingabezeile verläßt. Dadurch wird vermieden, daß das Programm mit einer Fehlermeldung abstürzt.

```
10 REM Eingabebegrenzung durch Fenster
20 PRINT CHR$(27);"M": REM Scrollen verhindert
40 PRINT "NAMEN?";
50 OPEN 1,0
60 WINDOW PEEK(236),PEEK(235),F,PEEK(235),1
70 INPUT#1,A$
80 CLOSE 1
90 WINDOW 0,0,F,24
```

Anstelle der Konstanten F sollten Sie beim 40-Zeichen-Bildschirm den Wert 39 und beim 80-Zeichen-Bildschirm den Wert 79 einsetzen.

In Adresse 236 finden Sie den Wert der augenblicklichen Spalte und in Adresse 235 den der Zeile. Durch diese Form des WINDOW-Befehls wird der Text automatisch hinter die zuletzt ausgegebenen Zeichen geschrieben.

2.5.6. PRINT AT mit Windows

Schon für den C-64 wurden in allen möglichen Fachzeitschriften und Büchern Möglichkeiten angeboten, den PRINT AT-Befehl zu simulieren. Diese waren mal kürzer und mal wesentlich länger. Der 128er hat hierfür einen eigenen Befehl, nämlich CHAR. Doch nicht immer ist dieser Befehl ideal. Man kann zum Beispiel die Länge der Anweisung nicht bestimmen. Auch könnten sich Steuerzeichen in dem auszudruckenden String befinden. Auf einem genau konstruierten Bild kann dies dazu führen, daß das Bild seine wohlgestaltete Form verliert und das Chaos ausbricht. Hätten Sie in derselben Situation einen WINDOW-Befehl benutzt, wären Sie aus dem Schneider:

```
10 REM PRINT AT durch Windows
20 INPUT "ZEILE";ZE
30 INPUT "SPALTE";SP
40 INPUT "LAENGE";LA
50 F=PEEK(231)
50 PRINT CHR$(27);"M": REM Scrolling verhindern
60 WINDOW SP,ZE,SP+LA,ZEILE
60 PRINT "HALLO"
70 WINDOW 0,0,F,24
```

Auch in diesem Programm gibt F die Zeilenlänge des gerade benutzten Bildschirms wieder. F wird automatisch vom Programm abgefragt, damit es zu keinen Komplikationen kommen kann.

2.5.7. Löschen eines Teil-Bildschirmes

Ein Teil des Window-Befehls, auf den wir bis jetzt noch überhaupt nicht eingegangen sind, ist die Möglichkeit, den Fensterinhalt zu löschen. Das kann man auf zwei Arten bewerkstelligen:

```
1. 10 WINDOW 10,10,20,20
   20 PRINT "(CLR-HOME)";

2. 10 WINDOW 10,10,20,20,1
```

Der Nutzen dürfte klar sein. Die nötige Übersicht, für die wir den Window-Befehl schon gelobt haben, wird sofort gewährleistet. Wir möchten jetzt die Funktion des Window-Befehls etwas mißbrauchen. Mit dem Befehl wollen wir nämlich ein Stück des Bildschirms löschen, um den Platz anderweitig zu nutzen und dies, ohne gleich den gesamten Bildschirminhalt zu verlieren. Nehmen wir an, Sie wollen die untere Bildschirmhälfte des 40-Zeichen-Bildschirms löschen. Das dazugehörige Programm würde folgendermaßen aussehen:

```
10 WINDOW 0, 12, 39, 24 , 1
20 WINDOW 0,  0, 39, 24(, 0)
```

Natürlich können Sie diese Zeilen auch im Direktmodus verwenden. Schreiben Sie dazu die Befehle in der unteren Bildschirmhälfte durch den Doppelpunkt getrennt in eine Zeile. So werden auch diese beiden Befehle gleich mitgelöscht (wie praktisch).

2.5.8. Window-Inhalt sichern

Bitte stellen Sie sich folgendes vor:
Sie befinden sich mitten in einem Textverarbeitungsprogramm und haben den ganzen Bildschirm vollgeschrieben und wollen diesen Text nun speichern. Also gehen Sie ins Menue. Dieses Menue erscheint als Fenster auf dem Bildschirm. Bei guter Software erscheint der Text unter dem Fenster bei Verlassen desselben wieder auf dem Bildschirm.
Der WINDOW-Befehl im 128 beinhaltet diese Funktion leider nicht.
Entfernt man das Fenster wie oben angegeben wieder, bleibt auf dem Bildschirm noch der Inhalt des Windows zurück, der alte Inhalt ist hoffnungslos verloren. Was machen, wenn man nach dem "fensterln" genau denselben Text wieder auf dem Bildschirm haben möchte? Durch ein einfaches BASIC-Programm kann man diese Aufgabe auf dem 40-Zeichen-Bildschirm simulieren:

(Listing 2.5.8.a)

Die interessanten Teile des Programms finden Sie ab Programmzeile 60000:
In Zeile 60010 wird ein Feld mit dem Namen X() definiert. Dabei ist die Feldgröße von der Größe des Windows abhängig. Wollen Sie dieses Programm in Ihre eigenen Programme mit aufnehmen, dürfen Sie auf keinen Fall ein Feld mit dem selben Namen verwenden. Ebenso sollten Sie keine Variablennamen benutzen, die in dieser Routine vorkommen. Müssen Sie nur die Koordinaten für ein Fenster übergeben, können Sie diese Werte direkt in die Schleife anstelle der Variablen schreiben. Bei mehreren verschiedenen Windows ist die Übergabe durch Variablen praktischer.

Wie alle BASIC-Programme, so hat auch dieses einen kleinen Nachteil. Es ist zu langsam. Aber zum Glück gibt es ja noch die Maschinensprache, die uns bei diesem Problem schon oft geholfen hat.
Daher das gleiche noch einmal in Maschinensprache:

```
1400  A5 E7     LDA $E7       ; rechte Grenze des
                                Fensters.
1402  18        CLC
1403  E5 E6     SBC $E6       ; minus linke Grenze
1405  85 FF     STA $FF       ; = Länge des Fensters
1407  E6 FF     INC $FF       ; Länge = Länge + 1
1409  A9 04     LDA #$04      ; Bildschirmanfang
140B  85 FC     STA $FC       ; speichern
140D  A2 00     LDX #$00
140F  8A        TXA
1410  E4 E5     CPX $E5       ; $E5 = 1.Window-Zeile?
1412  F0 0B     BEQ $141F     ; ja, dann $141F.
1414  E8        INX
1415  18        CLC
1416  69 28     ADC #$28      ; nächste Zeile
1418  90 F6     BCC $1410     ; noch nicht alle
141A  E6 FC     INC $FC       ; Erhöht Zähler Hi-Byte.
141C  4C 10 14  JMP $1410
141F  18        CLC
1420  65 E6     ADC $E6       ; Fensteranfang Low
1422  90 02     BCC $1426
1424  E6 FC     INC $FC       ; Hi-Byte um 1 erhöhen
1426  85 FB     STA $FB       ; Low-Byte speichern
1428  85 FD     STA $FD       ; in $FB und in $FD.
142A  A5 FC     LDA $FC       ; High-Byte
142C  69 F7     ADC #$11      ; 17 dazu
142E  85 FE     STA $FE       ; und in High-Byte des
                                2. Zählers
1430  A0 FF     LDY #$FF
1432  C8        INY
1433  AD 6B 14  LDA $146B     ; Lesen oder Schreiben?
1436  D0 1A     BNE $1452     ; schreiben
1438  B1 FB     LDA ($FB),Y   ; Zeichen vom Bildschirm
143A  91 FD     STA ($FD),Y   ; speichern
143C  C4 FF     CPY $FF       ; Y-Register = Länge?
143E  D0 F2     BNE $1432     ; ungleich, also weiter
1440  E8        INX
1441  E4 E4     CPX $E4       ; X-Register = untere
                                Grenze?
```

```
1443  90 14     BCC $1459     ; kleiner
1445  AD 6B 14  LDA $146B     ; Geschrieben oder
                                gelesen?
1448  D0 04     BNE $144E     ; Geschrieben, nächste-
                                mal lesen
144A  EE 6B 14  INC $146B
144D  60        RTS           ; zurück ins BASIC
144E  CE 6B 14  DEC $146B
1451  60        RTS           ; zurück ins BASIC
1452  B1 FD     LDA ($FD),Y   ; Zeichen aus Speicher
1454    91  FB      STA ($FB),Y      ;  schreiben  auf
Bildschirm
1456  4C 3C 14  JMP $143C
1459  A5 FB     LDA $FB       ; Lo-Byte in den Akku
145B  18        CLC
145C  69 28     ADC #$28      ; Nächste Zeile
145E  90 04     BCC $1464     ; Kein Übertrag
1460  E6 FC     INC $FC       ; erhöhe Hi-Byte um 1
1462  E6 FE     INC $FE       ; erhöhe Hi-Byte des
                                zweiten Zählers um 1
1464  85 FB     STA $FB       ; Lo-Byte in 1. Zähler
1466  85 FD     STA $FD       ; Lo-Byte in 2. Zähler
1468  4C 30 14  JMP $1430
146B  00        BRK           ; Byte für Lesen (0)
                                oder Schreiben (1)
```

Nun kommt der BASIC-Lader. Ein Beispielprogramm für die Verwendung der Routine wurde schon implementiert. Zuerst wird das Maschinensprache-Programm an die richtige Stelle gespeichert. Das dauert ein paar Sekunden. Danach wird das Testprogramm gestartet.

(Programm 2.5.8.b)

Die eigentliche Routine beginnt erst in Zeile 180. Dieser Teil wird auch zuerst angesprungen. Bis 180 befindet sich ein Beispielprogramm.
Das Fenster, aus dem gelesen bzw. in das der Text geschrieben werden soll, muß immer aktiviert sein, wenn die

Routine angesprungen wird, da die Routine die Eckpunkte übernimmt.

2.5.9. Simulation mehrerer Fenster

Mit dem BASIC-Programm aus dem Kapitel "Window-Inhalt sichern" kann man mehrere Fenster simulieren. Für diesen Zweck muß es allerdings etwas abgeändert werden.

(Programm 2.5.9.a)

Nach dem Starten wird zuerst ein Bildschirm aufgebaut. Danach werden vierzig Fenster erzeugt und beschrieben. Anschließend werden die Windows der Reihe nach gelöscht. Als Ergebnis erhalten Sie auf dem Bildschirm wieder das Anfangsbild.
Dieses Programm soll lediglich das System erklären, mit dem man mehrere Windows benutzen kann. Um es auf Ihre eigenen Programme zu übertragen, müssen Sie nur einzelne Programmteile übernehmen.
Das Programm verarbeitet 40 Windows. Dieses ist durch den großen Variablen-Speicher des C-128 möglich. Trotzdem sollten Sie nicht zu großzügig über diesen Platz verfügen. Im Programm wurde das Feld X() mit 40*1004 Bytes angegeben. Die 40 steht für die Anzahl der Fenster. Benutzen Sie in Ihren eigenen Programmen weniger, so ändern Sie diese Zahl entsprechend um.

Es ist möglich, daß sich ein Fenster über den ganzen Bildschirm ausdehnt. In diesem Fall würden tausend Bytes für den Bildschirminhalt und vier für die Fensterkoordinaten benötigt, das ergibt 1004. Ist das größte von Ihnen verwendete Fenster kleiner, so können Sie diese Zahl ebenfalls verkleinern, wodurch viel Speicherplatz gespart wird.

2.6. Sprite-Handling

Ein sehr guter Beweis für die Qualität des neuen BASICs V7.0 sind zweifellos die zahlreichen Befehle zur Sprite-Behandlung. Angefangen von Befehlen, die die Sprite-Position abfragen, bis zu einem Sprite-Generator ist alles vertreten. Mit diesen Befehlen wird die Sprite-Programmierung erst attraktiv. Die Programmierung von Sprites auf dem C-64 ist dagegen katastrophal. Auf dem C-64 gibt es keinen einzigen Sprite-Befehl. Alle Werte müssen in den Speicher gepoked werden, was das gute Prinzip von Sprites gründlich vermieste.

Sprites finden hauptsächlich in Spielen Verwendung. Sie ziehen dann als Raumschiffe oder kleine Männchen über den Bildschirm. Man kann die Sprites aber auch fest auf dem Bildschirm stehen lassen und dort statt Figuren Zeichen abbilden, wie das beispielsweise im Kapitel 3 "Echtzeit-Uhr auf dem C-128" der Fall ist.
Auch das folgende Programm ist für diesen Zweck gedacht. Es kopiert ein beliebiges Zeichen aus dem Zeichengenerator in eins der acht Sprites. Das Sprite kann man natürlich danach noch beliebig verändern. Auf diese Weise kann man, basierend auf den alten Buchstaben, eine eigene Schrift entwickeln.

(Programm 2.6.a)

Zuerst muß der Bildschirmcode des Zeichens eingeben werden. Nachdem das Zeichen in ein Sprite kopiert wurde, springt der Computer automatisch in den eingebauten Sprite-Generator. Um das Zeichen zu ändern, stehen Ihnen die bekannten Befehle des Sprite-Generators zur Verfügung.
Doch mit einem Zeichen mit einer Matrix von 8*8 Punkten kann man noch nicht sehr viel anfangen. Außerdem ist der größte Teil des Sprites noch nicht ausgenutzt. Daher verdoppelt das zweite Programm die Größe des Sprites. Dadurch kann jetzt auch der Multicolor-Modus der Sprites genutzt werden. So könnte man beispielsweise den Sprite mit einem Schatten

unterlegen. Eine andere Möglichkeit wäre, die Ecken der Buchstaben noch etwas abzurunden.

(Programm 2.6.b)

2.6.1. Design im Listing

Auch mit den vorzüglichen Befehlen ist eine Frage noch nicht vollkommen geklärt: Wie speichert man den Inhalt eines Sprites ab? Für dieses Problem sind die beiden folgenden Programme gedacht. Das erste Programm liest Daten in ein Sprite. Das besondere daran ist, daß in den DATA-Zeilen keine Daten, sondern Strings abgespeichert werden, wodurch die Erstellung von Sprites natürlich bedeutend schneller und einfacher von sich geht. Außerdem können keine falsche Daten auftreten und schon definierte Sprites können schnell umdefiniert werden.
Leider läuft das Programm nicht so schnell wie ein gewöhnlicher BASIC-Lader ab; das sollte man jedoch in Kauf nehmen können.

(Programm 2.6.1.a)

Wie Sie aus dem Listing ersehen können, wird ein gesetztes Bit eines Sprites als Sternchen, ein nicht gesetzes als Punkt dargestellt.

Während die Daten eingelesen werden, wird die Taktfrequenz verdoppelt. Dadurch kann der Rechner doppelt so schnell rechnen. Dabei kann aber auch der 40-Zeichen-Bildschirm nicht mehr ordnungsgemäß vom Computer bedient werden und so kommt es zu einem Flimmern darauf. Sobald die Berechnungen beendet worden sind, wird wieder auf die normale Taktfrequenz umgeschaltet und das Bild auf dem 40-Zeichen-Bildschirm ist wieder zu entziffern. Nun müßte über diesen Bildschirm auch ein Sprite schweben.

Damit Sie nicht jedes Programm erst in diese Form der Daten bringen müssen, haben wir ein weiteres Programm geschrieben,

das die Daten aus dem Sprite-Speicher liest. Die DATA-Zeilen werden dabei automatisch vom Computer erstellt:

(Programm 2.6.1.b)

Sobald der Computer die Zeilen erstellt hat, können alle Zeilen außer den DATA-Zeilen mit einem DELETE-Befehl gelöscht werden. Danach sollten Sie das Programm DESIGN 1 und Ihr eigenes Programm vor die DATA-Zeilen schreiben.

2.6.2. Komfortable Sprite-Editierung

Nachdem dem Sprite-Generator ein Sprite definiert worden ist, möchte man es oft etwas verändern.
So haben Sie z. B. ein Raumschiff definiert und möchten nun das "Negativ" davon haben. Statt alle Daten umständlich umzurechnen, können Sie nun das folgende Programm benutzen. Aber es kann noch mehr: spiegeln an beiden Achsen und sogar noch drehen!
Doch zuerst einmal das Listing:

(Programm 2.6.2.a)

Während die Werte für die Sprites umgerechnet werden, fängt der 40-Zeichen-Bildschirm wieder an zu flimmern. Auch hier haben wir die Taktfrequenz erhöht. Nach jeder Funktion dieses Programms wird die Frequenz wieder zurückgesetzt, und das neu entstandene Sprite erscheint auf dem Bildschirm. Nach dem Drücken einer Taste geht es wieder im Menü weiter.

Hier eine Kurzbeschreibung der Funktionen:

1. Spiegeln
Das Sprite wird an der 11. Zeile des Sprites gespiegelt. Dadurch wird die oberste Zeile zur untersten usw. Die Punkte an sich wechseln jedoch nicht Ihre Seite.

2. Spiegeln an der Längsachse
Gespiegelt wird diesmal zwischen der 12. und 13. Spalte.

Bei beiden Spiegelfunktionen kann der Multicolor-Modus berücksichtigt werden. Allerdings ändern sich bei der zweiten Funktion die Farben.

3. Drehen um 180 Grad
Diese Funktion springt nacheinander die beiden Spiegelfunktionen an. Wenn ein Sprite um 180 Grad gedreht werden soll, sollte man keinesfalls zweimal um 90 oder 270 Grad drehen, da bei diesen beiden Funktionen etwas vom Sprite verloren geht.

4. Sprite ansehen
Mit dieser Funktion kann ein Sprite angesehen werden, ohne daß irgendwelche Veränderungen daran getätigt werden.

5. Invertieren
Durch diese Funktion wird ein nicht gesetztes Bit gesetzt und ein gesetztes Bit gelöscht. Dadurch entsteht das "Negativ" zu einem Sprite. Der Multicolor-Modus wird nicht extra berücksichtigt, man kann aber auch Multicolor-Sprites so behandeln. Allerdings wird das Ergebnis immer im Einfarb-Modus ausgegeben. Durch zweimaliges invertieren erhält man wieder das Ausgangsbild.

6. Drehen um 90 Grad
Diese Routine springt zuerst das Unterprogramm "Drehen um 270 Grad" an und spiegelt anschließend an der Längsachse.

7. Drehen um 270 Grad
Diese Routine dreht das Sprite um 90 Grad gegen den Uhrzeigersinn. Ein Punkt in der linken unteren Ecke erscheint nun unten rechts. Ein Punkt oben links ist nun unten rechts zu finden. Den rechten oberen Punkt finden Sie nicht mehr, da ein Sprite 24*21 Punkte hat und durch die Drehung pro Zeile drei Bits entfallen.

Noch ein Tip: Machen Sie sich mit Hilfe des Befehls "C" des Sprite-Generators eine Kopie, damit Sie das Original nicht zerstören!

3. NÜTZLICHE PROGRAMME

3. Nützliche Programme

****

Auf den folgenden Seiten finden Sie einige Programme, die Ihnen bei der täglichen Arbeit mit dem Computer helfen werden. Wir haben versucht, diese Programme einfach und übersichtlich zu gestalten, damit Sie sie für eigene Zwecke weiterentwickeln oder umändern können. Viel Spaß!

3.1. Fehlerbehandlung

"Jeder Anfang ist schwer" heißt ein sehr bekanntes und oft benutztes Sprichwort. Murphy würde sagen: "Wenn etwas schiefgehen kann, so wird es schiefgehen."
Das betrifft natürlich auch - oder besser besonders - den Programmierer. In der Anfangsphase fragt man sich oft: "Wo soll denn jetzt schon wieder der Fehler liegen?" Selbst fortgeschrittene Programmierer, die den Vorteil haben, die Fehlermeldungen zu kennen (an fortgeschrittene Programmierer: wie lange haben Sie im Handbuch die Fehlermeldungen aufgeschlagen?), sitzen oft einige Zeit vor fehlerhaften Zeilen.

Um all diesen Leuten zu helfen, gibt es in einigen BASIC-Versionen die Help-Funktion, bei einigen Computern auch über eine Taste anzusprechen. Die Besitzer des C-128 befinden sich in der glücklichen Lage, auch so eine Funktion zu besitzen, sogar eine sehr praktische. Statt nur die fehlerhafte Zeile anzuzeigen, wird der mutmaßliche Fehler innerhalb dieser Zeile sogar noch revers dargestellt (auf dem 40-Zeichen-Bildschirm) oder unterstrichen (auf dem 80-Zeichen-Bildschirm). Der Befehl kann auch innerhalb von BASIC-Programmen benutzt werden.
Doch das ist noch nicht alles. In der BASIC-Version 7.0 steckt noch einiges mehr. So gibt es zwar nicht den Befehl "ON ERROR GOTO", aber etwas ähnliches: TRAP. Gibt man diesen Befehl zu Anfang einer Zeile gefolgt von einer Zeilennummer an, so wird beim Auftreten eines Fehlers zu der angegebenen

Zeile gesprungen. Weiterhin gibt es den Befehl ERR$. Dieser Befehl hat das Format eines String-Arrays. Zusammen mit der Variablen ER erfüllt er folgende Funktion: In ER ist die Nummer des aufgetretenen Fehlers gespeichert, und in ERR$ kann mit PRINT ERR$(ER) der dazugehörige Fehlerstring ausgegeben werden. Dann gibt es auch noch die Variable EL. In ihr wird die Zeilennummer, in der der Fehler aufgetreten ist, abgespeichert.
Doch damit ist der Befehlsvorrat zur Fehlerbehandlung noch lange nicht erschöpft. Wurde bei einem BASIC-Programmm durch einen TRAP-Befehl zu einer bestimmten Zeile gesprungen, so bestehen drei Möglichkeiten, das Programm danach weiterzuführen. Das geschieht mittels des RESUME-Befehls. Gibt man einfach nur RESUME an, so versucht der Computer das Programm erneut in der Zeile fortzufahren, in der der Fehler auftrat. Daß das häufig zu einer Endlosschleife führt, werden Sie sich wohl vorstellen können. Als zweites existiert die Möglichkeit, eine Zeilennummer hinter dem RESUME-Befehl anzugeben. Das Programm wird dann in dieser Zeile fortgeführt. Schließlich könne Sie noch NEXT hinter den RESUME-Befehl schreiben. Dann wird das Programm direkt hinter dem fehlerhaften Befehl fortgeführt.

Wenn man schon einmal von Fehlersuche spricht, muß man auch noch TRON und TROFF erwähnen. Mit diesen Befehlen wird die TRACE-Funktion eingeschaltet. Dadurch wird beim Programmablauf immer die Zeile angezeigt, die gerade abgearbeitet wird.

Zeigen wir nun einige Anwendungen dieser Befehle:

1. Für Anfänger

Bevor Sie ein eigenes Programm eingeben, sollten Sie folgende Zeilen eintippen:

```
10 TRAP 1000
20 :
30 PRINT: PRIND
```

```
40 PRINT CHR$(147)"(C) 1985 BY WERNER"
50 FOR T=1 TO 10: PRINT: NEXT Y
60 PRINT"O. K."
70 END
80 :
1000 REM FEHLEREINSPRUNG
1010 PRINT"(YEL 1)"ERR$(ER)" ERROR IN "EL;
1020 HELP
1030 PRINT"(CRSR DOWN)(CYN)'W'EITER ODER 'E'NDE?"
1040 GET KEY A$
1050 IF A$="E" THEN END
1060 IF A$="W" THEN RESUME NEXT
1070 GOTO 1040
```

Taucht nun ein Fehler beim Ablaufen des Programms auf (wie z. B. in Zeile 30), so wird zur Zeile 1000 gesprungen. Dort wird erst einmal die Art des Fehlers und dann die Zeilennummer, in der er aufgetreten ist, angezeigt. Das ist nur das, was auch der Computer machen würde. Dann wird die fehlerhafte Zeile selbst angezeigt und zwar mit dem markierten Fehler (geschieht alles durch den HELP-Befehl). Nun können Sie entscheiden, ob Sie das Programm weiterlaufen lassen wollen oder es unterbrechen, um den Fehler zu berichtigen.
Aber es geht noch viel "anfängerfreundlicher". Durch die mitgegebene Fehlernummer kann man individuell zu jedem Fehler eine "Fehlersuchroutine" ablaufen lassen (so könnte z. B. der Fehler auf deutsch erklärt werden etc.). Da dieses Programm nicht schwer zu schreiben ist, jedoch einen etwas größeren Umfang haben würde, haben wir uns entschlossen, kein Beispiel dafür hier aufzulisten.
Ein Tip für Maschinensprache-Programmierer: So ein Programm läßt sich auch in Maschinensprache schreiben. Hier ist es sogar noch brauchbarer, da sich damit restlos alle Fehlermeldungen auskosten lassen (die Fehler ILLEGAL DIRECT oder CAN'T CONTINUE z. B. würden wohl kaum innerhalb von Programmen auftreten). Realisieren können Sie das über den Vektor $0300/$0301. In dieser Adresse steht die Adresse, zu der der Computer beim Auftreten eines Fehlers (sowie bei END) springt. Das X-Register enthält bei einem Sprung über

diesen Vektor die Nummer des Fehlers. Wurde das Programm ohne einen Fehler beendet, so enthält das X-Register den Wert $80. Die Fehlernummern können folgende Werte annehmen:

Nummer (hex.)	Nummer (dez.)	Fehler
$01	1	TOO MANY FILES
$02	2	FILE OPEN
$03	3	FILE NOT OPEN
$04	4	FILE NOT FOUND
$05	5	DEVICE NOT PRESENT
$06	6	NOT INPUT FILE
$07	7	NOT OUTPUT FILE
$08	8	MISSING FILENAME
$09	9	ILLEGAL DEVICE NUMBER
$0A	10	NEXT WITHOUT FOR
$0B	11	SYNTAX
$0C	12	RETURN WITHOUT GOSUB
$0D	13	OUT OF DATA
$0E	14	ILLEGAL QUANTITY
$0F	15	OVERFLOW
$10	16	OUT OF MEMORY
$11	17	UNDEF'D STATEMENT
$12	18	BAD SUBSCRIPT
$13	19	REDIM'D ARRAY
$14	20	DIVISION BY ZERO
$15	21	ILLEGAL DIRECT
$16	22	TYPE MISMATCH
$17	23	STRING TOO LONG
$18	24	FILE DATA
$19	25	FORMULA TOO COMPLEX
$1A	26	CAN'T CONTINUE
$1B	27	UNDEF'D FUNCTION
$1C	28	VERIFY
$1D	29	LOAD
$1E	30	BREAK
$1F	31	CAN'T RESUME
$20	32	LOOP NOT FOUND

$21	:	33	: LOOP WITHOUT DO
$22	:	34	: DIRECT MODE ONLY
$23	:	35	: NO GRAPHICS AREA
$24	:	36	: BAD DISK
$25	:	37	: BEND NOT FOUND
$26	:	38	: LINE NUMBER TOO LARGE
$27	:	39	: UNRESOLVED REFERENCE
$28	:	40	: UNIMPLEMENTED COMMAND
$29	:	41	: FILE READ

2. Die Verwendung von "RESUME"

Vorhin wurde erwähnt, daß bei der Verwendung von RESUME ohne Zeilennummer und ohne NEXT eine Endlosschleife entstehen kann. Dieses ist natürlich nicht im Sinne des Erfinders. Trotzdem ist diese Art der Verwendung von RESUME brauchbar. Ein Beispiel zeigt folgendes Programm:

```
1 TRAP 1000
10 CATALOG
20 END
1000 IF ER=5 THEN PRINT CHR$(19);"BITTE FLOPPY EINSCHALTEN":
RESUME
1010 END
```

Schalten Sie nun Ihre Floppy aus und starten Sie das Programm. Es wird versucht, das Inhaltsverzeichnis auszugeben. Da die Floppy nicht angeschaltet ist, wird mit dem Fehler "DEVICE NOT PRESENT" zu der Zeile 1000 gesprungen. Da dieser Fehler die Nummer 5 hat, wird die Meldung "BITTE FLOPPY ANSCHALTEN!" ausgegeben. Durch den RESUME-Befehl geschieht das solange, bis die Floppy wirklich angeschaltet wird.

Nun noch ein Beispiel für die Verwendung von RESUME mit Zeilennummer:

```
1 TRAP 1000
10 INPUT"NAME DES FILES";NA$
```

```
20 INPUT"GERAETE-ADRESSE";GA
30 LOAD NA$,GA
40 END
1000 IF ER = 4 THEN PRINT"EIN FILE MIT DIESEM NAMEN
EXISTIERT NICHT": RESUME 10
1010 IF ER = 5 THEN PRINT"BITTE FLOPPY ANSCHALTEN": RESUME
1020 IF ER = 8 THEN PRINT"BITTE EINEN FILENAMEN ANGEBEN":
RESUME 10
1030 IF ER = 9 THEN PRINT"GERAETEADRESSE VON 8 - 15
waehlen": RESUME 20
1040 END
```

Wie Sie sehen, können Sie mit diesen Befehlen erreichen, daß ein Programm so gut wie absturzsicher wird, d. h., ein Benutzer kann durch falsche Angaben keine Fehlfunktion auslösen. Ein gutes Beispiel ist das oben stehende Programm: Mit wenigen Zeilen kann erreicht werden, daß das Programm bei nicht eingeschaltetem Floppy-Laufwerk oder Drucker nicht abstürzt, sondern nur eine Meldung ausgibt. Durch diese Funktion könnte so manches Textverarbeitungsprogramm erheblich verbessert werden.

3.2. Listing-Konverter

Mit seinem letzten Geld kauft man sich eine Computerzeitschrift, eilt nach Hause, schaltet seinen Rechner ein, schlägt die Seite mit dem Programm, das man unbedingt haben wollte, auf und fängt an abzutippen. Doch plötzlich hält man inne. Nein, es nicht Schmerz in den Fingern oder der Gedanke, daß das Programm doch nicht so gut ist, wie man dachte. Nein, man kann nur wieder einmal die Steuerzeichen nicht erkennen. War es nun ein reverser Kreis oder ein reverses Herzchen? Oder auch wenn man es erkennt: Was stellt denn nun ein reverser Pfeil da? Entnervt gibt man das Abtippen auf und kauft sich entweder eine Listingdiskette oder nie wieder diese Zeitschrift.
Doch zum Glück geht es auch anders. Einige Zeitschriften sind in letzter Zeit dazu übergegangen, ihre Programme vor dem Abdrucken durch einen Konverter zu schicken. Dann steht statt einem reversen Herzchen plötzlich "(CLR HOME)" auf dem Papier.
Nun sollen auch Sie in den Genuß kommen, Ihre Programme so ausdrucken zu können. Doch dieser Konverter hat noch andere Vorteile. Dazu erst einmal wieder eine kleine Geschichte aus dem Alltag: Unser Computerzeitschriftenkäufer von eben ist in den seltenen Besitz eines lesbaren Listings gekommen, sogar die reversen Steuerzeichen lassen sich gut erkennen. Doch beim Eintippen erleidet er schon wieder Frust. Diesmal konnte man ihn vor sich hinmurmeln hören: Sind es 10 oder 11 Leerzeichen? Ist das hier 39 oder 40 mal Cursor rechts? Arrrghh...
Doch hier tritt wieder der Konverter in Aktion: Es wird bei jedem Leerzeichen und bei jedem Steuerzeichen innerhalb von Strings die Anzahl mit angegeben.
Und noch etwas: Zwischen jeden Befehl kommt ein Leerzeichen. Das ist besonders hilfreich für Anfänger. Statt FORES=1TOP:W=PEEK(ES)ANDNOT5OR8:NEXTES steht dann da FOR ES = 1 TO P: W = PEEK (ES) AND NOT 5 OR 8: NEXT ES. Das ist zwar länger, aber auch übersichtlicher.
Der hier abgedruckte Konverter hat noch zwei Nebeneffekte:

Um es mit den Leerzeichen nicht zu übertreiben, wird auch wirklich nur ein Leerzeichen zwischen Befehle gesetzt. Hatten Sie eigentlich mehr vorgesehen, so werden diese gelöscht. Außerdem werden Befehle zum Anfang einer Zeile eingerückt, Variablen oder Doppelpunkte bleiben, werden jedoch ganz normal mit nur einem Leerschritt zwischen Zeilennummer und dem betreffenden Zeichen ausgegeben. Dadurch erhält das Listing einen interessant "Character". Wer schon einmal auf einem Apple-Rechner programmiert hat, kennt dieses Listing-Design schon. Auch hier werden Befehle eingerückt, Variablen jedoch nicht.

Nun zu dem Programm. Tippen Sie es erst einmal ab und speichern Sie es dann.

(listing 3.2.a)

Nun starten Sie es. Nach kurzer Wartezeit werden einige Eingaben von Ihnen verlangt. Drücken Sie bei der ersten Frage "B" für Bildschirm und bei der zweiten "G" für Großschrift. Dann geben Sie als Namen "KONVERTER" ein. Dieser Name ist nicht für irgendein Laden von Disk, sondern nur eine Überschrift zum Listing.
Sodann wird, falls Sie das Listing richtig abgetippt haben, der Konverter konvertiert auf dem Bildschirm erscheinen.
Schalten Sie nach korrektem Ablauf nun Ihren Drucker ein und probieren Sie das ganze mit Drucker. Hier werden die Strings, die statt der Steuerzeichen eingesetzt werden, natürlich nicht wie auf dem Bildschirm revers ausgegeben, denn sonst hätten wir wieder die gleiche Situation wie zu Anfang.

Wollen Sie nun andere Programme konvertiert ausgeben, so gehen Sie folgendermaßen vor:

a) Laden Sie zuerst Ihr auszugebendes Programm in den Speicher.
b) Setzen Sie nun den BASIC-Anfang hinter dieses Programm. Das geht folgendermaßen: Zeiger $2D/$2E hinter das Programm setzen (Wert schätzen oder über die FRE-Funktion) und eine

Null in die Adresse vor der Adresse, auf die dieser Zeiger zeigt.
Wenn Sie sich nicht auf Schätzen verlassen wollen, so können Sie auch folgende Routine verwenden:

```
1300 A0 00    LDY #$00
1302 E6 2D    INC $2D
1304 D0 02    BNE $1308
1306 E6 2E    INC $2E
1308 B1 2D    LDA ($2D),Y
130A F0 F6    BNE $1302
130C E6 2D    INC $2D
130E D0 02    BNE $1312
1310 E6 2E    INC $2E
1312 B1 2D    LDA ($2D),Y
1314 F0 EC    BNE $1302
1316 E6 2D    INC $2D
1318 F0 02    BNE $131C
131A E6 2E    INC $2E
131C B1 2D    LDA ($2D),Y
131E F0 E2    BNE $1302
1320 E6 2D    INC $2D
1322 F0 02    BNE $1326
1324 E6 2E    INC $2E
1326 60       RTS
```

Speichern Sie am besten diese Routine auf die Diskette mit dem Konverter-Programm ab.
c) Geben Sie NEW ein (auch bei dem Maschinensprache-Programm).
d) Laden Sie nun den Konverter und starten Sie ihn wie gehabt.

Wir haben uns für diese Möglichkeit entschieden. Man hätte natürlich den Konverter auch so schreiben können, daß erst der Konverter und dann das auszugebende Programm geladen wird. Das wäre aber kein Unterschied gewesen. Als weitere Möglichkeit haben wir in Betracht gezogen, daß sich der Konverter das Programm direkt von Diskette holt. So hätte man nur den Konverter laden müssen. Diese Version haben wir

aber für zu langsam und zu schwer verständlich gehalten. Sie hat allerdings auch einen Vorteil: Das auszugebende Programm kann länger sein (da bei der jetzigen Version BASIC-Speicher durch den Konverter verloren geht).

Noch ein paar Erklärungen zum Konverter:
Das Programm geht folgendermaßen vor: Es fängt beim BASIC-Anfang an, jedes Byte einzeln zu holen und zu interpretieren. Stößt es auf eine Null, so testet es die nächsten beiden Bytes. Sind diese auch Null, so ist das Ende des auszugebenden Programms erreicht, und damit wird auch der Konverter beendet. Andere Zeichen werden erst einmal darauf getestet, ob es Token sind. Dabei muß natürlich unterschieden werden, ob das Byte innerhalb oder außerhalb eines Strings liegt. Bei den Token sind noch zwei Sonderfälle zu berücksichtigen: Wie Sie aus der Token-Tabelle im Anhang ersehen können, sind zwei Werte doppelt belegt ($CE und $FE). Stößt der Computer auf einen dieser beiden Werte, so holt er auch noch das nächste Byte und dann erst den dazugehörigen Befehl. Diese beiden "Spezial-Token" müssen beim Konverter natürlich noch besonders berücksichtigt werden.

Die Strings, die statt Farbsteuerzeichen ausgegeben werden, entsprechen den Aufdrucken auf den entsprechenden Tasten.

Wir haben uns bemüht, das Programm übersichtlich und damit gut verstehbar zu schreiben. Sie können es so leicht Ihren eigenen Bedürfnissen anpassen. So könnte es Sie z. B. stören, daß für jedes Leerzeichen innerhalb von Strings eine Meldung ausgegeben wird. Denkbar ist z. B., daß nur ab drei Leerzeichen die Meldung ausgegeben wird.
Zum Schluß noch ein paar Worte zu den Variablen:

DR : Ist gleich 0, wenn das Listing nicht auf dem Drucker ausgegeben werden soll, sonst 1.
FL : Ist 1, wenn gerade die Zeichen hinter einem REM-Befehl ausgegeben werden. Dadurch bleibt die Anzahl von Leerzeichen

hinter REM-Befehlen mit der von Ihnen angegebenen Anzahl identisch.
AN : Adresse des gerade gelesenen Bytes.
WE : Inhalt dieser Adresse (also das Byte, von dem hier dauernd die Rede ist).
FF : Ist 0, wenn das Byte außerhalb eines Strings und 1, wenn es innerhalb eines Strings gelesen wurde.
ZA : Ein Zähler, der die Anzahl der gleichen Zeichen hintereinander zählt. Hier (in den Zeilen 6040 - 6070) müssen Sie eingreifen, wenn Sie erst ab drei Leerzeichen den String "(SPACE)" ausgeben wollen.

TK$(255): Enthält die Befehls-Strings zu den einzelnen Token (durch diese Methode werden zwar ganz schön viel Bytes verschwendet, aber wir haben es ja...)
FE$(29) : Enthält den Befehls-String zu Befehlen, deren erstes Byte der Wert $FE ist.
CE$(8) : Das gleiche für Befehle, die mit $CE beginnen.
SZ$(159): Enthält die auszugebenden Strings für die einzelnen Codes der Steuerzeichen.

3.3. OLD

Wem ist es noch nicht vorgekommen, daß man ein mühselig eingegebenes Programm durch ein unvorsichtiges NEW gelöscht hat? Dann hilft alles nicht mehr.
Wie wir garantiert alle wissen, stimmt das nicht. Vom C-64 her dürften wohl die zahlreichen OLD-Routinen bekannt sein, die alle darauf beruhen, daß der Computer bei NEW das BASIC-Programm nicht löscht, sondern nur zwei Nullen an den Anfang schreibt. Die OLD-Routinen berechneten dann einfach die gelöschten Bytes neu. Diese fehlenden Bytes zeigten im Low/High-Byte-Verfahren auf die nächste Zeilennummer (genauer gesagt auf die Adresse, an der die nächste Zeilennummer im Speicher anfängt). Man mußte also nur diesen Anfang herausfinden, diese Adresse in Low- und High-Byte zerlegen und in die gelöschten Bytes schreiben. Danach mußte der Anfang der Variablen hinter das Programm gesetzt werden, da sonst das Programm sich bei der Definition von Variablen selbst gelöscht hätte.
Diese OLD-Routinen werden jedoch erfahrungsgemäß ziemlich lang und so oft fehlerhaft. Es gibt jedoch noch eine viel einfachere Möglichkeit:

a) Man schreibt in das zweite fehlende Byte eine beliebige Zahl (natürlich ungleich Null). Am besten ist es, wenn man das High-Byte des BASIC-Anfanges wählt.
b) Nun ruft man eine BASIC-Interpreter-Routine auf. Diese Routine bindet die BASIC-Programmzeilen neu und wird z. B. benutzt, wenn eine neue Programmzeile eingefügt wird. Dadurch verändern sich ja auch die ganzen Vektoren (Links genannt) und müssen neu berechnet werden. Genau das wird dem Rechner jetzt hier auch "vorgegaukelt". Der Link der ersten Programmzeile, der durch die beiden Nullen gelöscht wurde, wird einfach irgendwo in den Speicher gesetzt und dann neu berechnet.
c) Nun wird der Anfang der Variablen hinter das Ende des Programms gesetzt.

Damit ist das gelöschte Programm wieder betriebsbereit.

Beim C-128 geht es nun noch einfacher. Der Schritt c) kann einfach weggelassen werden, da die Variablen in der RAM-Bank 1 abgespeichert werden, und so nicht mit dem Programm in der RAM-Bank 0 kollidieren.

Man kann also OLD ganz einfach mit folgender Zeile durchführen:

POKE PEEK(45)+256*PEEK(46)+1,28: SYS DEC("4F4F")

Die Adressen 45/46 zeigen auf den BASIC-Anfang. Da das zweite Byte angesprochen werden soll, muß noch eins dazuaddiert werden. In diese Adresse wird nun das High-Byte des normalen BASIC-Anfanges geschrieben. Dann wird die Routine aufgerufen, die die Links neu berechnet.

Nun das ganze noch einmal in Maschinensprache:

```
0B00 A0 01    LDY #$01        ; Offset 1
0B02 A9 1C    LDA #$1C        ; 28 für BASIC-RAM-Anfang
0B04 91 2D    STA ($2D),Y     ; und ins High-Byte des Links
0B06 20 4F 4F JSR $4F4F       ; Links neu berechnen
0B09 60       RTS             ; Und Ende
```

Sie können die Routine natürlich beliebig in den Speicher setzen, nicht nur bei $0B00.

OLD funktioniert natürlich auch, wenn Sie den BASIC-Anfang verändert haben. Sogar nach RESET wird die Routine nicht verzweifeln.
Eine Ausnahme gibt es allerdings: wenn der BASIC-Anfang hochgesetzt wurde und dann RESET durchgeführt wird. Dann sucht OLD natürlich am normalen BASIC-Anfang nach einem Programm und nicht an dem, den Sie eingestellt hatten. Das macht aber nichts, denn hier brauchen Sie sowieso kein OLD. Stellen Sie einfach den BASIC-Anfang wieder auf die vorher eingestellten Werte und ihr Programm wird wieder zum Vorschein kommen.

3.4. Musik nebenbei

Schon beim Eintippen von Texten auf der Schreibmaschine hat es sich bewährt, nebenbei bestimmte Melodien zu hören. Es ist nur sehr ungünstig, wenn Sie nichts in der Nähe haben, was Musik von sich gibt. Und nicht jeder ist der geborene Sänger. Mit einer Schreibmaschine wären Sie jetzt aufgeschmissen. Nicht aber mit Ihrem guten alten C-128. Man kann nämlich mit einem Programm erreichen, daß, während man wie gewöhnlich mit dem Computer arbeitet, ein Musikstück ertönt. Voraussetzung ist natürlich, Sie haben auch etwas, das Ihnen den Ton ausgibt, beispielsweise den Lautsprecher eines angeschlossenen Fernsehers oder eine Stereoanlage etc..
Zum BASIC-Lader läßt sich sagen, daß er sehr lang ist. Der Grund dafür ist, daß das Lied 99 Noten spielt, bevor es wieder von vorne anfängt. Pro Note müssen die Frequenz in High- und Low-Byte sowie der Notenwert angegeben werden. Das ergibt insgesamt 297 Musikdaten, die natürlich alle untergebracht werden wollen. Bis einschließlich Zeile 160 befindet sich das Maschinensprache-Programm samt Daten. Anschließend werden die Werte für die Musik gesetzt.

(Programm 3.4.a)

Damit Ihnen die Musik nicht langweilig wird, sollten Sie selbst noch andere Stücke in dieses Programm einbauen. Ein Lied kann bis zu 255 Noten haben. Jede Note benötigt wie schon erwähnt drei Werte:

1. Ton-Frequenz Low-Byte
2. Ton-Frequenz High-Byte
3. Notenwert

Sie können alle Tonwerte selbst wählen. Wir haben bei dieser Routine beispielsweise für den Ton G die Frequenz 3338 genommen. Dann ergibt das als Low-Byte 10 und 13 als High-Byte. Als Notenwert haben wir zum Beispiel für eine Viertelnote den Wert 10 angenommen. Bei eigenen Musikstücken

muß natürlich die Zahl der Noten in Zeile 230 entsprechend geändert werden. Wollen Sie die Wellenform ändern, fügen Sie bitte diesen Poke-Befehl an das Programmende:

```
POKE 7226, Wellenform
```

Die Werte haben dabei diese Funktion:

```
17  Dreieck
33  Sägezahn
65  Rechteck
129 Rauschen
```

Dieses Programm benutzt nur die dritte Stimme. Sie können es auch neben BASIC-Programmen laufen lassen, solange diese nur die erste und zweite Stimme benutzen. Wollen Sie das Programm in Ihre eigenen Maschinensprache-Programme mit einbauen, so sollten Sie sich den Programmverlauf genau anschauen. Das Programm verbiegt den Vektor für den IRQ, so daß bei jedem IRQ-Ansprung zuerst die kleine Routine angesprungen wird. Außerdem benutzt die Routine mehrere Adressen als Zeiger und Speicher:

```
    $1400 (5120) Routine, um IRQ-Vektor zu verbiegen
 bis $1415 (5141)
    $1416 (5142) Anfang der Musik-IRQ-Routine
 bis $144E (5198)
    $14F1 (5361) Zählt Notenwert, wenn 1CF1 gleich Null,
                 dann wird die nächste Note geholt
    $14F3 (5363) Zähler für Noten
    $1500 (5376) Speicher für Low-Byte der Noten
 bis $15FF (5631)
    $1600 (5632) Speicher für High-Byte der Noten
 bis $16FF (5887)     $1700   (5888)   Speicher   für    die
Notenwerte
bis $17FF (6143)
```

Mit dieser Liste der Speicherinhalte verstehen Sie das Programm wesentlich einfacher. Das Programm kann auch anstatt mit dem BASIC-Lader mit diesem nun folgenden

Maschinensprache-Listing und dem eingebauten Monitor eingegeben werden:

```
1400  78        SEI            ; verhindert Interrupt
1401  A9 16     LDA #$16       ; setzt IRQ-Vektor auf
1403  8D 14 03  STA $0314      ; den Anfang der
1406  A9 14     LDA #$14       ; Musik-Routine
1408  8015 03 STA $0315
140B  58        CLI            ; erlaubt Interrupt
140C  A0 00     LDY #$00
140E  8C F3 14  STY $14F3
1411  CB        INY
1412  8C F1 14  STY $14F1
1415  60        RTS            ; Zurück ins BASIC.
1416  CE F1 14  DEC 14F1       ; erniedrigt Zähler
1419  D0 31     BNE 144C       ; Zähler nicht Null,
                               ; normal weiter
141B  48        PHA            ; Akku auf Stack
141C  8D 00 FF  STA $FF00      ; schaltet I/O an
141F  8D 12 D4  STA $D412      ; Wellenform auf 0
1422  8A        TXA            ; X-register in Akku
1423  48        PHA            ; Akku auf Stack
1424  AE F3 14  LDX $14F3      ; Nummer für neue Note
1427  BD 00 15  LDA $1500,X    ; Holt Note Lo-Byte
142A  8D 0E D4  STA $D40E      ; setzt Lo-Byte
142D  BD 00 16  LDA $1600,X    ; Holt Hi-Byte
1430  8D 0F D4  STA $040F      ; setzt Hi-Byte
1433  BD 00 17  LDA $1700,X    ; holt Notenwert
1436  8D F1 14  STA $14F1      ; setzt Zähler
1439  A9 41     LDA #$41       ; lädt Wellenform
143B  80 12 D4  STA $D412      ; setzt Wellenform
143E  E8        INX            ; Notenzähler erhöhen
143F  38        SEC
1440  E0 63     CPX #$63       ; letzte Note ?
1442  90 02     BCC $1446      ; nein,dann $1C46
1444  A2 00     LDX #$00       ; Notenzähler auf 0
1446  8E F3 14  STX $14F3      ; Notenzähler speichern
1449  68        PLA            ; Akku vom Stack
144A  AA        TAX            ; X-Register = Akku
```

```
144B  68        PLA          ; Akku von Stack
144C  4C 65 FA  JMP $FA65    ; zum normalen IRQ
```

3.5. Echtzeituhr auf dem Commodore 128

Ohne Uhr ist man in der heutigen Zeit aufgeschmissen - man kommt nicht pünktlich zur Arbeit, zu Verabredungen und kann auch seine Leistungen nicht effektiv werten. Um mit dem Trend der Zeit zu gehen, wurden auch in Computer Uhren eingebaut. So kann man beim C-128 z. B. über TI$ die jeweilige Zeit erfahren (die natürlich einmal gesetzt werden muß). Doch TI$ hat einen Nachteil: Sie geht ganz schön ungenau. Kein Problem - es gibt noch eine genauere Uhr, die man aus diesem Grund auch "Echtzeit-Uhr" nennt. Diese Uhr wird über Zähler in einem der CIAs realisiert - in unserem Fall im CIA#1 (ab $DC00).

Doch schauen Sie sich die Uhr erst einmal an:

(Programm 3.5.a)

Die einzelnen Ziffern werden durch Sprites dargestellt. Aus diesem Grund besteht das Listing auch zu einem großen Teil aus Daten für diese Sprites.
Sie können das Programm aber auch ohne Sprites benutzen. In diesem Fall brauchen Sie das Programm nur ab Zeile 780 einzugeben. Zusätzlich entfallen die Zeilen 1090 - Ende. Danach muß nur noch die Zeile 1070 geändert werden, so daß die Uhrzeit auf den Bildschirm ausgegeben wird:

```
1070 DATA 104,234,234,153,255,  3,136
```

Als letztes müssen Sie die Prüfsumme noch verändern. Sie lautet bei dieser Variante "6314".

Nun noch das Maschinensprache-Listing:

```
1400  78        SEI          ; verhindert Interrupt
```

```
1401  A9 0D     LDA #$0D     ; setzt den IRQ-Vektor
1403  8D 14 03  STA $0314    ; auf den Anfang der
1406  A9 14     LDA #$14     ; Routine.
1408  8D 15 03  STA $0315
140B  58        CLI          ; erlaubt Tnterrupt
140C  60        RTS          ; zurück ins BASIC
140D  A0 03     LDY #$03
140F  AD 0B DC  LDA $DC0B    ; Uhrzeit Stunde
1412  29 1F     AND #$1F     ; nur die 5 ersten Bits
1414  4C 1A 14  JMP $141A
1417  B9 08 DC  LDA $DC08,Y  ; Uhrzeit
141A  29 F0     AND #$F0     ; nur Zehnerstelle
141C  4A        LSR          ; verschiebt Bit 4-7
141D  4A        LSR          ; nach Bit 0-3
141E  4A        LSR
141F  4A        LSR
1420  48        PHA          ; legt Zahl auf Stack
1421  B9 08 DC  LDA $DC08,Y  ; Uhrzeit
1424  29 0F     AND #$0F     ; nur Einerstelle
1426  48        PHA          ; legt Zeit auf Stack
1427  88        DEY
1428  D0 ED     BNE $1417
142A  AD 08 DC  LDA $DC08    ; Uhrzeit - Zehntel
1420  48        PHA          ; legt Zehntel auf Stack
142E  A0 07     LDY #$07
1430  68        PLA          ; holt Zahl vom Stapel
1431  18        CLC
1432  69 36     ADC #$36     ; addiert Anfang Sprites
1434  99 F7 07  STA $07F7,Y  ; Sprite auf Block
1437  88        DEY
1438  D0 F6     BNE 1430
143A  4C 65 FA  JMP $FA65    ; zum normalen IRQ
```

Bei der Variante, bei der keine Sprites verwandt werden, ändern sich nur zwei Zeilen:

```
1C32  69 30     ADC #$30     ; Zahl in ASCII umwandeln
1C34  99 FF 03  STA $03FF,Y  ; und auf Bildschirm
```

Wie Sie dem Listing vielleicht entnehmen konnten, wird die Uhr durch den IRQ gesetzt. Dadurch verschwindet sie auch sofort, wenn RUN-STOP/RESTORE gedrückt werden (der IRQ-Vektore wird wieder auf die normalen Werte gesetzt und die Uhr-Routine wird nicht mehr angesprungen).

3.6. Analoguhr

Um wirklich im Trend der Zeit zu bleiben, nun noch ein Programm, mit dem die Zeit dargestellt wird.
Nach dem Motto "Warum eigentlich immer digital?" geschieht die Anzeige diesmal analog - wie es die älteren Leser noch aus ihrer Kindheit her kennen werden (wer die Anzeige nicht lesen kann - bitte Großeltern fragen!).

Die Anzeige selbst wird über die Grafik-Befehle des C-128 auf den Bildschirm gebracht. Dadurch erscheint die Uhr - wie die vorhergehende - nur auf dem 40-Zeichen-Bildschirm.

Nun zu dem Programm: Es läßt sich in zwei Einzel-Programme unterteilen. Der erste Teil könnte Ihnen bekannt vorkommen. Es ist der gleiche Programmteil, der auch schon bei der vorigen Uhr zum Setzen der Uhr benutzt worden war; auch diese Uhr basiert also auf der eingebauten Hardware-Uhr Ihres Computers. Ab Zeile 150 beginnt der zweite Programmteil. Hier wird die Uhr gezeichnet, wobei Stunden- und Minutenzeiger wie gewöhnlich im Uhrkreis, der Sekundenzeiger jedoch am äußeren Rand als Strich erscheint. Jede Minute, wenn der Minuten- und der Stundenzeiger bewegt werden müssen, wird gleich der gesamte Bildschirm gelöscht und wieder ganz neu aufgebaut. Dieser Vorgang nimmt nicht sehr viel Zeit in Anspruch.
Die drei Zeiger werden alle mit dem CIRCLE-Befehl erstellt. Mit diesem Befehl läßt sich die Drehung der Zeiger ohne irgendwelche Formeln bewerkstelligen, was sehr viel Zeit spart.

(Programm 3.6.a)

3.7. LLIST

Oft kommt es vor, daß Listings von BASIC-Programmen schnell einmal auf den Drucker ausgegeben werden sollen. In den BASIC-Versionen, die auf den Commodore-Rechnern vertreten sind, geht das ziemlich umständlich:

```
OPEN 4,4: CMD 4: LIST: CLOSE 4
```

Bei besseren BASIC-Versionen existiert dafür ein Befehl: LLIST.

Beim C-64 hatte man einige Schwierigkeiten, diesen Befehl dem BASIC-Wortschatz dazuzufügen. Das lag daran, daß nach dem Listen zum BASIC-Warmstart gesprungen wurde (daher konnte man LIST auch nicht innerhalb von Programmen anwenden). Beim C-128 ist das etwas anderes. Hier wird nach LIST genauso wie nach PRINT oder ähnlichen Befehlen wieder in die Stelle des Programms zurückgesprungen, bei der der Befehl auftrat. Bei einem LLIST-Befehl muß man also nur die Ausgabe auf den Drucker umlenken, dann die LIST-Routine anspringen und schließlich die Ausgabe wieder auf den Bildschirm lenken.

Bei der nun folgenden LLIST-Routine wird die LIST-Routine nicht direkt am Anfang angesprungen, sondern schon etwas später. Das liegt daran, daß nicht geprüft werden muß, ab wann bzw. bis wann das Programm gelistet werden soll (es wird immer das ganze Programm auf den Drucker ausgegeben). Dafür muß aber auch ein Vektor gesetzt werden: $61 / $62 auf den Anfang der ersten Zeile, ab der gelistet werden soll (hier gleich dem BASIC-RAM-Anfang).

Nun das Assembler-Listing:

```
1A00 A9 04     LDA #$04        : Logische Dateinummer
1A02 AA        TAX             : Geräte-Adresse
1A03 A0 00     LDY #$00        : Sekundärnummer
1A05 20 BA FF  JSR $FFBA       : File-Parameter setzen
```

```
1A08 20 C0 FF JSR $FFC0      : Kernal-Routine OPEN
1A0B A2 04    LDX #$04       : Logische Dateinummer
1A0D 20 C9 FF JSR $FFC9      : Ausgabegerät setzen
1A10 A5 2D    LDA $2D        : BASIC-RAM-Anfang low
1A12 85 61    STA $61        : abspeichern
1A14 A5 2E    LDA $2E        : BASIC-RAM-Anfang high
1A16 85 62    STA $62        : abspeichern
1A18 20 E5 50 JSR $50E5      : LIST-Routine (alles Listen)
1A1B A9 04    LDA #$04       : Logische Dateinummer
1A1D 20 C3 FF JSR $FFC3      : Kernal-Routine CLOSE
1A20 A9 00    LDA #$00       : 0 = Bildschirm
1A22 85 BA    STA $BA        : Aktuelle Geräteadresse
1A24 20 C9 FF JSR $FFC9      : Ausgabegerät setzen
1A27 RTS                     : Und Rücksprung
```

3.8. Textverarbeitung Marke Eigenbau

Keine Angst! Wir haben nicht vor, jetzt auf langen Seiten Aufbau und Programmierung einer eigenen Textverarbeitung zu erläutern; dies wäre sicherlich nicht Sinn eines "Tips & Tricks" Buches.
Gemäß unserem Motto "Tips & Tricks" wollen wir Ihnen vielmehr eine einfache und ebenso verblüffende Alternative zu Textverarbeitungsprogrammen bieten. Oft kommt es vor, daß man mal kurz einen kleinen Text auf dem Drucker ausgeben möchte. Jetzt erst Textverarbeitung reinladen, warten, erstellen, drucken? Es geht auch einfacher, wenn man auf den Komfort moderner Textverarbeitung verzichten kann. Und das funktioniert so:

Sie geben den gewünschten Text einfach als Programm ein. Tippen Sie dazu

```
AUTO 10
```

ein. Jetzt geht's los. Geben Sie als erste Zeilennummer:

```
10 ...
```

ein und schreiben Sie munter vor sich hin. Sie sollten allerdings darauf achten, daß Sie pro Zeilennummer nur soviel Zeichen eingeben, wie auch pro Zeile ausgedruckt werden sollen / können. Nach jeder vollen Zeile muß also die RETURN-Taste gedrückt werden (kleiner Tribut an diese einfache Textverarbeitung, man gewöhnt sich dran!).
Der Anfang Ihrer "Textverarbeitung" könnte also beispielsweise so aussehen:

```
10 DIES IST DIE NEUE TEXTVERARBEITUNG, DIE MIT EINFACHEN
20 MITTELN BEACHTLICHE ERFOLGE ERZIELT.
30 WENN SIE WISSEN WOLLEN, WIE ES WEITERGEHT...
   (...)
```

Alles schön und gut, aber wie kommt der Text jetzt fein säuberlich aufs Papier? Nichts einfacher als das. Geben Sie folgende Zeile im Direktmode ein, wenn Sie den Text vollständig eingegeben haben:

```
POKE 24,37:OPEN 4,4:CMD 4:LIST
```

und anschließend:

```
PRINT#4:CLOSE 4
```

Wie Sie sehen, sehen Sie gar nichts mehr - zumindest von den störenden Zeilennummern!
Sie können übrigens auch auf diese Weise erstellte Texte als normales Programm auf Disk oder Kassette absaven und so für späteren Gebrauch retten!

Wenn Sie sich mit der hier vorgestellten Möglichkeit der Textverarbeitung mal etwas näher beschäftigen, werden Ihnen sicher noch viele Verbesserungsvorschläge in den Sinn kommen. Und eines Tages stehen Sie dann doch noch vor Ihrer eigenen professionellen Textverarbeitung!

Die Adresse 24

Mit dieser Adresse läßt sich aber noch mehr machen. Zum Beispiel:

```
POKE 24,37:LIST
```

listet ein Programm ohne Zeilennummern auf dem Bildschirm auf.

Tritt einmal in einem Ihrer Programme ein "unheilbarer" "?FORMULATOOCOMPLEXERROR"auf,dannversuchenSieeinmal:

```
POKE 24,27
```

Dieser POKE wirkt oft Wunder!

Andere Werte als 27 in Adresse 24 bewirken mitunter recht eigentümliche Veränderungen der BASIC-Listings, ohne jedoch die Lauffähigkeit der Programme einzuschränken. Probieren Sie doch einfach ein paar Werte aus!
Nach einer jeglichen Fehlermeldung wird der Inhalt der Adresse 24 neu gesetzt.

3.9. Modifiziertes INPUT

Das neue V7.0-BASIC besitzt eigentlich alle Eingabe-Befehle, die man normalerweise benötigt: GET, GETKEY und INPUT.
Leider hat letztere Anweisung einen kleinen Schönheitsfehler: Es erscheint ein Fragezeichen auf dem Bildschirm, als Kennzeichen für Abfragebereitschaft. Was aber tun in folgendem Fall?

```
10 INPUT "GEBEN SIE IHREN NAMEN EIN!";N$
```

Hinter das Ausrufungszeichen setzt der Computer dreist und eigenmächtig noch ein Fragezeichen. Sowas kann man gar nicht gebrauchen. Abhilfe schafft da folgender Trick:

Eröffnen Sie doch den Bildschirm als Datei! Anschließend

können Sie dann ohne das berüchtigte INPUT-Fragezeichen arbeiten:

```
10 OPEN 1,0     (Bildschirm als Datei)
20 INPUT#1,A$   (Abfrage ohne Fragezeichen)
30 CLOSE 1
```

Soll die INPUT-Anweisung an einer bestimmten Stelle des Bildschirmes erscheinen, funktioniert das so:

```
20 CHAR,Spalte,Zeile
25 PRINT" ";:
27 INPUT$1,A$
```

Die Cursorpositionierung mit CHAR wird erst nach einer PRINT-Anweisung ausgeführt.

3.10. Signalton

Nicht erst bei neueren Schreibmaschinen erscheint, wenn eine bestimmte Spalte überschritten wird, ein akkustisches Warnsignal. Dies hat auch seinen guten Grund. Auf diese Weise braucht man den Blick nicht von der Vorlage oder von der Taste abzuwenden, um nachzusehen, in welcher Spalte man gerade herumtippt.

Diese Aufgabe wollen wir nun einem pflichtbewußten Maschinensprache-Programm übergeben:

```
142D  A9 FF     LDA #$FF      ; Musikwerte werden
142F  8D 06 D4  STA $D406
1432  8D 18 D4  STA $D418
1435  A9 09     LDA #$09
1437  8D 05 D4  STA $D405
143A  78        SEI           ; verhindert IRQ
143B  A9 47     LDA #$47      ; setzt IRQ-Vektor neu.
143D  8D 14 03  STA $0314
1440  A0 14     LDA #$14
```

```
1442  8D 15 03  STA $0315
1445  58        CLI          ; erlaubt IRQ
1446  60        RTS          ; Zurück ins BASIC
1447  48        PHA          ; Akku auf Stack
1448  A5 E7     LDA $E7      ; lädt rechte Grenze
144A  E5 Ec     SBC $EC      ; zieht Crsr.-Spalte ab
144C  C9 02     CMP #$02     ; gleich zwei ?
144E  D0 11     BNE $1461    ; nein, dann $1461
1450  CD 80 14  CMP $1480    ; Spalte beim letzten
1453  F0 0F     BEQ $1464    ; IRQ auch 2, dann 1464
1455  8D 80 14  STA $1480    ; 2 abspeichern
1458  A9 A0     LDA #$A0     ; Ton ausgeben
145A  8D 01 D4  STA $D401
145D  A9 21     LDA #21
145F  D0 05     BNE $1466
1461  8D 80 14  STA $1480    ; Spalte abspeichern
1464  A9 00     LDA #$00     ; Töne ausschalten
1466  8D 01 D4  STA $D401
1469  8D 00 D4  STA $D400
146C  8D 04 D4  STA $D404    ; Wellenform setzen
146F  68        PLA          ; Akku vom Stack
1470  4C 65 FA  JMP $FA65    ; Sprung zum IRQ

1480        Zeiger für vorige Spalte.
```

Genau zwei Zeichen vor Ende einer Zeile kann der geschulte Lauscher einen Ton vernehmen. Das ist nicht abhängig von der Zeichenlänge des jeweils angeschlossenen Monitors oder Fernsehers. Auch bei Fenstern kann man einen Ton hören, sobald die vorletzte Spalte erreicht wird. Das Programm beachtet nämlich die Adresse für die rechte Grenze eines Windows. Von diesem Wert wird die augenblickliche Cursorposition abgezogen. Ist das Ergebnis zufällig zwei, so läßt der Computer seinen Freudenton entweichen.

(Programm 3.10.a)

Dieses Programm muß aber nicht erst bei der vorletzten Spalte einen Ton ausgeben. Alternativ können Sie auch jeden

anderen beliebigen Wert einsetzen, wenn Sie schon vorher oder gar später gewarnt werden wollen. Dieser POKE-Befehl ist nur an das Programm anzuhängen. Anstatt des Parameters N müssen Sie natürlich die Spaltennummer einfügen. Achten Sie dadrauf, die Spalten von rechts nach links zu zählen.

```
POKE 5197, N
```

Dieses Programm eignet sich außer für Textverarbeitung natürlich auch ausgezeichnet zum Eingeben von Daten. Für diese beiden Gebiete eignet sich noch ein anderes Programm ausgezeichnet. Es ist das Programm "Tastaturpiep" aus dem Kapitel "Tastaturbelegung". Um zu verhindern, daß Sie auf eines dieser Programme verzichten müssen, haben wir diese Programme so geschrieben, daß nur wenige Veränderungen im Programm vorgenommen werden müssen.
Nehmen Sie sich als erstes das Tastaturpiep-Listing vor. Bei diesem Listing entfällt die Zeile 210. Ab Zeile 210 müssen Sie stattdessen die Daten nehmen, die mit Zeile 210 anfangen. In dem nun entstandenen Listing müssen wir nur noch zwei Werte ändern: die Prüfsumme und die Daten-Anzahl. Ändern Sie also Zeile 20 wie folgt:

```
20 FOR I=5120 TO 5234
```

Um die Prüfsumme richtig zu stellen, ändern Sie die Zeile bitte wie folgt ab:

```
70 IF S<>13738 THEN BEGIN
```

4. SOFTWARE-SCHUTZ AUF DEM COMMODORE 128

4. Software-Schutz auf dem Commodore 128

****

Große Software-Häuser stecken viel Geld und vor allem Hirnschmalz in die Entwicklung immer neuer und besserer Kopierschutz-Möglichkeiten. Es ist kaum vorstellbar, wieviel Schaden durch illegale Kopiererei angerichtet wird. Nicht zuletzt wird auch die Mühe des Programmautors nicht mehr entsprechend honoriert.
Bei Ihren eigenen Programmen wird es aller Wahrscheinlichkeit nach nicht darum gehen, daß Ihre Programme unerlaubt kopiert werden. Vielmehr ist es eine Gemeinheit, auf einem eigenen Programm ein fremdes Copyright zu finden. Außerdem möchte man nicht unbedingt jedem seine Programmiertricks und -kniffe verraten. Leider gibt es da für BASIC-Programmierer einen schrecklichen Gegner: der LIST-Befehl. Liefert er bei der Programmerstellung noch nützliche Dienste, so bietet er jetzt jedem "Programmeinbrecher" eine willkommene Hintertür!
Es gibt glücklicherweise aber einige Methoden, diese Hintertür zu verstellen, sprich: den LIST-Befehl außer Kraft zu setzen. Eine Auswahl dieser Methoden finden Sie auf den folgenden Seiten. Teilweise können die hier vorgestellten Schutzroutinen mit geringfügigen Änderungen im C64-Mode verwendet werden, da der BASIC-Programm-Aufbau derselbe ist. Die Listschutz-Pokes gelten jedoch ausschließlich für den C128!

4.1. Schutz durch Doppelpunkte

Unser erklärtes Ziel ist es, jedem Programmeinbrecher den Zugriff auf besondere, schützenswerte Zeilen zu versagen. Diese Arbeit erledigt die folgende kleine BASIC-Routine für Sie. Ihre Arbeit besteht darin, alle Zeilen in Ihrem Programm zu kennzeichnen, damit sie die Routine findet. Stellen Sie vor jede Zeile, deren Inhalt geschützt werden soll, fünf Doppelpunkte! Aus der Zeile:

```
450 PRINT "(C) 1985 BY 'THE TEAM'"
```

würde die gekennzeichnete Zeile:

```
450 :::::PRINT "(C) 1985 BY 'THE TEAM'"
```

Wenn Sie schließlich damit fertig sind, tippen Sie das folgende Listing zu Ihrem Programm:

(Programm 4.1.a)

Die Schutzroutine wird mit:

```
RUN 60000
```

aktiviert. Sie können nun sehen, welche Programmzeile die Routine gerade überprüft (PASS). Trifft sie auf eine markierte Zeile, so wird diese geschützt und ein Stern als Zeichen auf den Bildschirm gebracht. Sobald sich die Routine mit "FERTIG !" meldet, können Sie sie mit "DELETE 60000-60110" wieder löschen; sie wird jetzt nicht mehr benötigt. Listen Sie ruhig das Programm auf! Zwar sind noch die Zeilennummern lesbar, der Inhalt der markierten Zeilen ist jedoch verschwunden! Der Listschutz wurde im BASIC-Speicher selbst realisiert, weswegen er automatisch mit auf Disk oder Kassette gesaved wird.

Das Programm:

```
60000 : Der Anfang des BASIC-Programms wird ermittelt und
        in der Variablen LI hinterlegt. Wollen Sie das Pro-
        gramm im C64-Mode benutzen, so müssen Sie lediglich
        die in dieser Zeile vorkommende Speicherstelle
        45 durch 43 ersetzen!
60010 : Um nicht jedes Byte des Speichers durchsuchen zu
        müssen, wird hier der Zeilen-Link bestimmt, der die
        Anfangsadresse der nächsten Zeile enthält. Dadurch
        muß nicht der gesamte Inhalt einer Zeile durchleuch-
        tet werden; die Routine arbeitet schneller!
60020 : Das Programmende ist erreicht, wenn der aktuelle
```

Link LI=0 ist. Das Programm ist fertig.

60030 : Die aktuelle Zeilennummer wird ausgegeben. Sie befindet sich in den auf den Link folgenden beiden Adressen.

60040 : Die Anzahl der geschützten Zeilen in X$ wird ausgegeben.

60050 : Die Basis der neuen Zeilenanfangsberechnung ist gleich dem aktuellen Link LI.

60060 : Die ersten fünf Speicherstellen jeder Zeile werden auf die fünf Doppelpunkte hin untersucht. Ist bereits eine Stelle ungleich, wird das Flag FL gesetzt und X erhält den Wert 8, damit die Schleife sofort verlassen werden kann.

60080 :

60090 : Ist FL=1 dann wird das Flag zurückgesetzt und die nächste Zeile gesucht.

60100 : Die Zeile wird geschützt, indem der erste der fünf Doppelpunkte durch 0 ersetzt wird. Findet die LIST-Routine diese 0, interpretiert sie sie als Zeilenende, was zur Folge hat, daß der Inhalt der Zeile unsichtbar bleibt. Im Programmlauf werden die nachfolgenden vier Doppelpunkte jedoch als Koppelglieder interpretiert und so der Programmlauf problemlos durchgeführt.

4.2. Zeilennummern-Roulette

Charakteristischstes Merkmal eines BASIC-Programms sind wohl die durchlaufenden Programmzeilen-Nummern. In ihrer Reihenfolge wird das Programm abgearbeitet, sie dienen dem Programmierer gewissermaßen als Leitfaden. Es ist nicht möglich, Zeilennummern größer als 64000 zu wählen. Zeilennummern stehen immer in numerischer Reihenfolge.
Dieses Idyll an Ordnung und Zufriedenheit wollen wir nun etwas durcheinander bringen. Dabei leistet die folgende Routine wertvolle Dienste. Tippen Sie sie zu Ihrem Programm!

(Programm 4.2.a)

Sicherlich werden Ihnen die ersten Zeilen dieser Routine von der vorangegangenen her bekannt vorkommen. Sie erfüllen hier denselben Zweck. Starten Sie nun die Routine. Wieder wird die Zeilennummer der Zeile angegeben, die gerade bearbeitet wird. Sie haben jetzt drei Möglichkeiten zur Auswahl:

```
1. (A)ENDERN
2. (W)EITER
3. (E)NDE
```

"E" beendet das Programm; "W" sucht die nächste Zeile. Wenn Sie jedoch "A" als Programmpunkt wählen, können Sie die aktuelle Zeilennummern Ihren Wünschen entsprechend verändern. Die neue Zeilennummer muß lediglich im Bereich zwischen 0 und 65535 liegen. Folgendes sollten Sie allerdings beim Ändern der Zeilennummern unbedingt beachten, um den Programmablauf nicht zu gefährden:

a) Sprungbefehle wie GOTO oder GOSUB können veränderte Zeilen nicht anspringen.
b) Alle Zeilennummern, die kleiner als eine vorangegangene sind, können nicht angesprungen werden.
c) Zeilennummern größer als 64000 können nicht mehr gelöscht werden.

Das Programm:

Das Prinzip dieser Routine ist recht simpel. Sie sucht nacheinander alle Zeilen eines BASIC-Programms. Soll die Zeilennummer einer Zeile verändert werden, wird die neue Zeilennummer zunächst in ZN gespeichert und dann in Zeile 60100 ins Lo/Hi-Byte-Format zerlegt. Anschließend werden die neuen Werte an Stelle der alten in den BASIC-Speicher gepoked.
Auch dieser Schutz wird automatisch mit auf Disk oder Kassette gesaved.

Übrigens - eine interessante Anwendung: Tja, was anfangen mit dieser netten Routine? Geht man auf Nummer sicher und will unter allen Umständen den korrekten Programmablauf

gewährleistet wissen, gibt es trotzdem eine nützliche Anwendung. Geben Sie folgende Zeile zu Ihrem Programm:

```
64000 REM (C) 1985 BY LOS DOS DESPERADOS
```

Anschließend starten Sie die Hilfsroutine. Wird Zeile 64000 angezeigt, so verändern Sie sie einfach in:

```
65535
```

Sie erzielen damit einen doppelten Effekt: Erstens befindet sich die Zeile mit der Zeilennummer 65535 wirklich am Programmende und stört so den Ablauf nicht. Zweitens wird es jedem schwerfallen, Ihr in der REM-Zeile abgelegtes Copyright zu löschen, da nur Zeilen mit einer Zeilennummer kleiner als 65000 gelöscht werden können (sofern man nicht über diese kleine Routine verfügt!).

4.3. Manipulation des Zeilen-Links

Die vorangegangenen Routinen bedienten sich des sogenannten Zeilen-Links, um den Anfang einer neuen Zeilennummer zu finden. Genau diesen Link werden wir jetzt verändern. So könnte man den Link einer Zeile auf die übernächste "verbiegen". Die Folge: Die dazwischenliegende Zeile erscheint nicht auf dem Bildschirm. Oder der Link wird auf eine vorangegangene Zeile gerichtet, so daß die betreffenden Zeilen endlos aufgelistet werden. Lassen Sie ruhig Ihrer Phantasie freien Lauf; im Normalfall wird nämlich der Programmlauf nicht gestört, so wild Sie auch den Link verändern! Wieder möchten wir Ihnen zu diesem Thema eine Hilfsroutine an die Hand geben:

(Programm 4.3.a)

Nach dem Start der Routine werden Sie nach der ersten und zweiten Zeilennummer gefragt. Anschließend sucht das Programm beide Zeilennummern und legt den Link der ersten Zeile auf den der zweiten. Alle Zeilen dazwischen bleiben also unsichtbar!

Es sei allerdings dazu gesagt: Sobald das Listing verändert wird, sei es durch das Hinzufügen einer neuen oder das Löschen einer alten Zeile, wird der Link vom Interpreter neu berechnet. Der Schutz wird also hinfällig. Zur Ehrenrettung sei jedoch hinzugefügt, daß Programmeinbrecher oft gar nicht erst ans Ändern gehen, wenn keine interessanten Zeilen aufgelistet werden!

4.4. Künstliche Steuerzeichen: ein kleiner Kobold

Als solchen kann man durchaus ein bestimmtes Steuerzeichen bezeichnen. Steuerzeichen werden Ihnen nicht unbekannt sein. Sie treten in BASIC-Programmen innerhalb von PRINT-Anweisungen sehr häufig auf; als Farb- oder Cursorsteuerzeichen zum Beispiel. Hier ist jedoch die Rede von einem recht eigenwilligen Steuerzeichen. Es läßt sich nur sehr schwer auf den Bildschirm bringen, deshalb haben wir uns eines kleinen Tricks bedient. Geben Sie bitte folgende Zeile im Direktmodus ein:

```
KEY
1,CHR$(27)+CHR$(79)+"(rvson)(SHIFT+M)(rvsoff)"+CHR$(34)+CHR$
(20)
```

Wir haben einfach eine Funktionstaste mit dem Steuerzeichen belegt (die vielen CHR$s haben lediglich den Zweck, vor Ausgabe des Steuerzeichens den Quote-Mode aus- und anschließend wieder einzuschalten!) : F1. Drücken Sie diese Taste einmal! Es erscheint auf dem Bildschirm ein dunkles Quadrat mit einem diagonalen Strich darin. Was dieses Zeichen vermag, wollen wir Ihnen im folgenden zeigen. Dazu finden Sie ein paar BASIC-Zeilen, die Sie direkt eingeben können. An Stelle eines * setzen Sie bitte das Steuerzeichen durch Betätigung von F1!

Der Kobold und PRINT

```
PRINT "So wirkt das kleine Ding*in PRINT-Anweisungen*!"
```

Innerhalb von PRINT-Anweisungen erfüllt es die Funktion eines Wagenrücklaufes (CR). Doch dies nur am Rande, denn es hat mit Software-Schutz nun wirklich nichts zu tun!

Der Kobold und REM

```
REM "*(crsrup)(rvson)Dies ist eine Testzeile
```

Hier wird die Sache schon interessanter! Das merkwürdige Zeichen bewirkt nämlich in Verbindung mit einem Anführungszeichen hinter einer REM-Anweisung, daß alle darauf folgenden Zeichen wie eine PRINT-Anweisung behandelt werden. Im Klartext: Alle so behandelten REM-Zeilen werden zu funktionsfähigen PRINT-Zeilen, die während des LIST-Vorganges ausgeführt werden!

Listings färben

Jetzt sind Ihnen wirklich keine Grenzen mehr gesetzt! Sie können so das Listing farbig gestalten...

```
1000 REM"*(ctrl+1bis8)
1000 REM"*(c= + 1bis8)
```

...das Listing auseinanderziehen...

```
1000 REM"*(crsrdown)(crsrdown)
```

oder einfach Zeilen maskieren:

```
1000 REM"*(crsrup)900 SYS (4096) CRACKED BYUNS
(täuscht ein gecracktes Maschinenprogramm vor)

1000 PRINT"(C) BY UNS":REM"*(crsrup)
(Zeile 1000 wird von nachfolgender Zeile sofort wieder
 überschrieben)
```

Viel Spaß mit dem kleinen Ungeheuer!

4.5. Schutz durch Pokes

Mit Pokes kann man so vieles machen, so auch hier. Da gibt es die Möglichkeit zu verhindern, daß ein Programm abgesaved, gelistet oder gestartet werden kann.
Leider haben alle diese Pokes einen Nachteil: Sie wirken erst nach dem Starten des Programms. In Verbindung mit einem Autostart können sie jedoch sehr wirkungsvoll sein.

4.5.1. Verhindern des LIST-Befehls

In der Page 3 existiert ein Zeiger, der auf die Adresse zeigt, zu der bei Eingabe des LIST-Befehls gesprungen wird. Dieser Zeiger hat die Adresse 774 / 775 ($0306 / $0307).
Verbiegen wir diesen Vektor einmal (geben Sie vorher NEW ein):

```
POKE 774,61: POKE 775,255
```

Geben Sie nun einmal "LIST" ein. Der Computer springt ganz normal zurück (da sich ja kein Programm im Speicher befindet). Geben Sie nun eine Programmzeile ein. Sollten Sie jetzt noch einmal "LIST" eingeben, so wird der Rechner zuerst die erste Zeilennummer ihres Programms ausgeben, und dann einen Reset durchführen.
Zur Erklärung: Der LIST-Vektor wurde auf die Reset-Routine gelegt. Da jedoch bei Eingabe des LIST-Befehls zuerst geprüft wird, ob sich überhaupt ein Programm im Speicher befindet, wurde zuerst kein Reset ausgeführt. Nachdem Sie ein Programm eingegeben hatten, sprang der Computer über den Zeiger 774 / 775 zu der (vermeintlichen) LIST-Routine. Da er dort jedoch die Reset-Routine vorfand, führte er einfach diese aus.

Versuchen Sie nun einmal den:

```
POKE 774,38: POKE 775,160
```

Dadurch zeigt der Vektor auf ein "RTS". Bei einem "LIST" werden nun nur die Zeilennummern ausgegeben. Das liegt daran, daß der Vektor 774 / 775 nicht wirklich auf die LIST-Routine zeigt, sondern nur auf eine Routine, die einen BASIC-Text in Zeichen umwandelt (z. B. die Token in Strings zurückverwandelt). Die List-Routine holt nun die jeweilige Zeilennummer, gibt sie aus und springt dann zu der "Umwandlung in Klartext"-Routine. Dort steht jedoch nur ein "RTS", und die nächste Zeilennummer wird geholt.

Wie Sie sehen, kann man mit diesem Zeiger viele schöne Dinge anstellen.

4.5.2. Ausschalten von RUN-STOP / RESTORE

Schon früh sind Leute auf die Idee gekommen, Code-Worte in ihre Programme einzubauen. Kein Problem für unterlaubte Eindringlinge - sie betätigten einfach RUN-STOP & RESTORE, verließen so das Programm und schauten einfach das Code-Wort nach.
Ganz so leicht geht es nun nicht mehr. Mit

```
POKE 808,PEEK(808) - 3
```

werden diese beide Tasten einfach ihrer Funktion beraubt. Nun kann man sie sogar noch abfragen:

```
10 POKE 808,PEEK(808) - 3
20 PRINT"DAS IST EIN TEST ";
30 GET A$: IF A$=CHR$(3) THEN PRINT: PRINT"*BREAK*"
40 GOTO 20
```

Der Phantasie sind hier keine Grenzen gesetzt.

Nur RESTORE kann übrigens mit

```
POKE 792,51: POKE 793,255
```

ausgeschaltet werden.

Nur RUN-STOP kann sogar in BASIC abgefangen werden:

```
10 TRAP 1000
20 PRINT"DAS IST EIN TEST ";
30 GOTO 20
1000 PRINT"*BREAK*"
1010 RESUME NEXT
```

Hierbei wird ausgenutzt, das die Fehlermeldung "BREAK", die ja bei Drücken der RUN-STOP-Taste auftritt, eine Fehlermeldung wie jede andere auch ist, und so wird einfach die eingebaute "ON ERROR GOTO"-Variante benutzt. Nähere Erklärungen dazu finden Sie im Kapitel 3: "Fehlerbehandlung". Allerdings: Ganz sicher ist diese Methode nicht (drücken Sie einmal mehrmals schnell hintereinander auf die RUN-STOP-Taste).

4.5.3. Verhindern von SAVE

Auch in der Page 3 existiert ein Vektor, der auf die SAVE-Routine zeigt. Dies ist der Zeiger 818 / 819 ($0332 / $0333). Mit diesem Vektor können Sie die üblichen Scherze treiben: ihn auf Reset zeigen lassen, ihn auf ein "RTS" setzen etc., etc...

4.6. Kopierschutz für Disk

Bis jetzt haben wir mehr oder weniger erfolgreich versucht, das Listen eines Programms zu unterbinden. Aber die Programme konnten immernoch munter kopiert werden. Das soll jetzt anders werden!

Vielleicht haben Sie schon mal was von der Methode gehört, Tracks auf Disk zu "zerstören". So läuft das Programm zwar ordnungsgemäß, ein Kopierprogramm stößt früher oder später aber auf die zerstörte Stelle und kann nicht weitermachen. So etwas ähnliches haben wir auch vor. Wir wollen einen READ ERROR auf Disk schreiben lassen.

Das folgende Programm erledigt diese Aufgabe. Es muß lediglich der gewünschte Track in der Variablen TR angegeben werden:

```
 0 REM READ ERROR AUF DISK
10 OPEN 1,8,15
20 OPEN 2,8,2,"#"
30 PRINT#1,"U1 2 0";TR;0
40 PRINT#1,"M-E"CHR$(163)CHR$(253)
50 END
```

Bevor Sie dieses Programm starten, vergewissern Sie sich, ob nicht eine wertvolle Diskette im Laufwerk schmort. Zum Ausprobieren sollten Sie zunächst eine Diskette nehmen, auf deren Inhalt Sie notfalls verzichten können!

4.7. Directory-Klauer

Oft benötigt man gar nicht so schwerwiegende Eingriffe wie oben beschrieben, um zu verhindern, daß Programme von Disk kopiert werden.
Das Directory, das Inhaltsverzeichnis der Floppy also, gibt verräterische Infos an potentielle Kopierer weiter. Diese

können mit ihm nach Programmen fahnden, die es zu Kopieren lohnen würde. Was tut man? Man läßt einfach das gesamte Directory verschwinden!

Zunächst dazu das entsprechende Listing:

```
10 REM DIRECTORY VERSCHWINDEN LASSEN
20 OPEN 1,8,3,"#"
30 OPEN 2,8,15,"B-P3,144"
40 PRINT#1,CHR$(20)CHR$(20)CHR$(20)CHR$(0)CHR$(0)CHR$(0)
50 PRINT#2,"U2:3,0,18"
60 PRINT#2,"I"
70 END
```

Auch hier sollten Sie nur Disketten verwenden, von denen Ihnen die Namen aller Programme bekannt sind; das Listen des Directory-Eintrages klappt ja nun nicht mehr.
Aber der Schutz geht hier noch etwas weiter: Es können nunmehr auch keine Programme gescratcht oder sonstwie verändert werden.

Das Prinzip:

...ist eigentlich ganz einfach. Das Directory wird wie ein BASIC-Programm behandelt und ausgegeben. Soll die Ausgabe also unterbunden werden, werden an den Anfang des Programms drei Nullen geschrieben.
Diese werden von der List-Routine als Programmende interpretiert.

5. DAS SYSTEM DER 1000 MÖGLICHKEITEN: ZEILEN EINFÜGEN.

5. Das System der 1000 Möglichkeiten: Zeilen einfügen

****

In den meisten Fällen eignen sich Programme nur für eine ganz bestimmte Anwendung, auf die sie speziell abgestimmt sind. Woran liegt das? Ganz offensichtlich ist das Hauptproblem, daß ein Programm (scheinbar) nicht mehr verändert werden kann, sobald es einmal gestartet ist.
Wir wollen Ihnen auf den folgenden Seiten zeigen, wie man doch Zeilen in ein "laufendes" Programm einfügt, welche Vorteile diesbezüglich gerade der Commodore 128 bietet und anhand einiger Beispielprogramme Anregungen geben, wie Sie diese neue Programmiertechnik sinnvoll nutzen können.

Schon oft wurde versucht, in laufende Programme Zeilen einzufügen, denn gelingt dies, bietet sich dem Programmierer ein riesiges Anwendungsspektrum. Stellen Sie sich doch bloß eine simple Rechenroutine vor, die zwei zuvor eingegebene Werte miteinander verknüpft! An sich nichts Tolles, werden Sie meinen. Hat man aber die Möglichkeit, nach dem Start des Programms noch Zeilen einzufügen, so könnte die Rechenvorschrift frei gewählt werden, nach der die beiden (oder mehrere) Werte miteinander verknüpft werden. Sie wird nachträglich als Zeile zum Programm zugefügt!
Von hier aus ist es auch kein großer Schritt mehr zu einem kompletten Programm-Generator, der nach den Anweisungen des Anwenders eigene BASIC-Programme generiert. Und damit sind die Möglichkeiten noch bei weitem nicht erschöpft!
Ja, WENN man Zeilen ins Programm einfügen könnte! Da wurden schon die kompliziertesten Verfahren entwickelt - mehr oder weniger erfolgreich. Die Methode, die wir Ihnen vorstellen möchten, besticht durch ihre einfache und zudem sehr zuverlässige Funktionsweise!

Programmzeilen in ein laufendes Programm einfügen ist unmöglich. Oder wenigstens äußerst kompliziert. Also muß das Programm unterbrochen werden, um eine Zeile einzufügen. Wie erreicht man aber, daß nach dem Programmstop die Zeile übernommen und das Programm erneut gestartet wird? Der

Schlüssel dazu ist der Tastaturpuffer. Bevor wir auf seine Wirkungsweise eingehen, eine kleine Kostprobe:

```
10 TP=842
20 POKE   TP,ASC("L")
30 POKE TP+1,ASC("I")
40 POKE TP+2,ASC("S")
50 POKE TP+3,ASC("T")
60 POKE TP+4,13
70 :
80 POKE 208,5
90 END
```

Sobald Sie dieses Programm starten, listet es sich selbst auf. Was ist passiert? In den Zeilen 20-50 wird das Wort "LIST" in den Tastaturpuffer geschrieben. Zeile 60 schreibt dahinter ein "RETURN" (Code 13). Schließlich wird dem Computer in Zeile 80 mitgeteilt, daß fünf Zeichen im Tastaturpuffer Ihrer Befreiung harren. Gleich nach dieser Nachricht versucht der Computer, die Zeichen aus ihrem Gefängnis zu befreien. Dies geht nur nicht, da der Computer mit der Abarbeitung des Programms beschäftigt ist. Er nimmt sich aber ganz fest vor, die Zeichen bei der nächsten Gelegenheit zu retten. Eine solche Gelegenheit ist immer dann gegeben, wenn der Computer auf eine Tastatureingabe wartet, bei INPUT oder GET also, oder wenn das Programm beendet ist. Genau dies geschieht: Das Programm wird in Zeile 90 beendet und der Computer gibt die im Tastaturpuffer "gefangenen" Zeichen aus. Da das Wort "LIST" mit RETURN abgeschlossen wurde, interpretiert der Rechner dieses gleich als Befehl und listet das Programm. Experimentieren Sie ruhig eine Weile mit diesem Programm! Lassen Sie beispielsweise einmal Zeile 60 weg, um das Ausgabeprinzip zu verstehen: Die Zeichen werden erst ausgegeben, wenn das Programm BEENDET ist. Oder schreiben Sie ein anderes Befehlswort in den Puffer (z.B. RUN, nicht vergessen: mit RETURN (13) abschließen !).

5.1. Anpaßbare Rechenroutine

Nun wollen wir darangehen und versuchen, eine ganze BASIC-Zeile in ein Programm einzufügen. Dies soll schrittweise geschehen, um den prinzipiellen Ablauf besser darstellen zu können. Als Beispiel wollen wir die oben bereits vorgeschlagene variable Rechenroutine realisieren!

```
10 PRINT "ANPASSBARE RECHENROUTINE"
20 PRINT:PRINT
30 PRINT "WIE LAUTET DIE RECHENVORSCHRIFT"
40 PRINT "(Z.B. Y=2*X+SQR(X/34.5)*INT(5.6*X^4)
50 INPUT V$
```

Dieser Programmteil dürfte ohne weiteres verständlich sein: Es wird die variable Rechenvorschrift erfragt und in V$ gespeichert. Die Vorschrift soll in unserem Beispiel das Format:

Y=....X...

haben. Es kann also eine beliebige Verknüpfung der Variablen X durchgeführt werden.
Wie soll aber jetzt die in V$ gespeicherte Vorschrift eingefügt werden?

```
60 PRINT CHR$(147);CHR$(145);CHR$(145);CHR$(145)
70 PRINT "1000";V$
80 PRINT "RUN 500"
```

Na, dämmert es? Starten Sie das Programm! Die Rechenvorschrift wird erfragt und anschließend in den oberen Teil des Bildschirms gebracht; vorgestellt ist die zukünftige Zeilennummer: 1000!

```
90 PRINT CHR$(19);
```

Jetzt dürfte das Prinzip vollends klar werden: Der Cursor blinkt auf der neu einzufügenden Zeile. Um sie ins Listing zu übernehmen, würde es genügen, ein paarmal die RETURN-Taste zu betätigen. So sehr es Sie auch in den

Fingern kitzelt, tun Sie es nicht! Dies soll ja das Programm erledigen:

```
100 FOR A=0 TO 3
110 POKE 842+A,13
120 NEXT A
130 POKE 208,4
140 END

500 PRINT CHR$(147)
510 INPUT "X-WERT";X
1000 REM HIER WIRD DIE VARIABLE RECHENVORSCHRIFT EINGEFÜGT
1010 PRINT "DAS ERGEBNIS LAUTET: ";Y
1020 END
```

Der Tastaturpuffer wird mit dem Code für RETURN (13) gefüllt. Anschließend wird dies wieder dem Computer mitgeteilt (lieber zu viel RETURNs als zu wenig!). Dann trifft das Programm in Zeile 140 auf ein END. Die RETURNs werden ausgegeben, die Zeile übernommen und mit "RUN 500" die Rechenroutine automatisch gestartet!
Sie sehen selbst, sooo unheimlich schwer ist es gar nicht! Versuchen Sie doch einmal, die Rechenroutine mit mehreren Variablen zu realisieren! Es ist nicht schwer: Lediglich die Rechenroutine ab Zeile 500 muß verändert werden. Das Einbinden der Rechenvorschrift kann ja dasselbe bleiben.

5.2. DATA-Generator

Wir möchten an dieser Stelle aber nicht aufhören. Vielmehr stellen wir Ihnen das nächste Programm als Anregung vor. Es handelt sich dabei um ein "echtes" Anwendungsprogramm: einen DATA-Generator. Falls Sie selbst in Maschinensprache programmieren, leistet er Ihnen nützliche Dienste. Aber auch sonst dürfte das Programm interessante Anregungen bieten, generiert es doch gleich mehrere Zeilen!

(Programm 5.2.a)

Zur Bedienung:

Der DATA-Generator generiert Ihnen einen BASIC-Loader. Sie müssen dem Programm lediglich Anfangs- und Endadresse der Maschinenroutine sowie Speicherkonfiguration (normalerweise 0) nennen. Außerdem verlangt das Programm Formateingaben wie erste Zeilennummer des neuen BASIC-Loaders, Zeilenabstand, etc.
Die erste Zeilennummer sollte größer als 1300 sein, da sonst der DATA-Generator überschrieben wird. Hat dieser seine Arbeit getan, kann er mit DELETE gelöscht werden. Ebenso lassen sich die Zeilennummern des generierten Loaders mit RENUMBER Ihren Wünschen anpassen.

Noch einmal kurz zusammengefaßt:

Adresse	(C64)	Bedeutung
208	198	Anzahl der Zeichen, die bei der nächsten Gelegenheit aus dem Tastarturpuffer ausgegeben werden sollen.
842-851	631-640	Tastaturpuffer (normal); in diese 10 Bytes können die Zeichen geschrieben werden, die ausgegeben werden sollen (13=RETURN).
2592	649	Maximale Größe des Tastaturpuffers, Sie ist auf 10 begrenzt, da nicht mehr Platz für den Puffer vorhanden ist.

Wichtig: Das Einfügen von Zeilen in ein lauffähiges Programm ist grundsätzlich beim Commodore 64 auch möglich. Jedoch ist dies beim C128 einfacher. Werden nämlich beim C64 neue Zeilen in ein Programm eingebunden, so werden dabei alle Variablen gelöscht. Dies ist beim C128 nicht der Fall, da

dieser Rechner für die Variablen eine getrennte Speicherbank (Bank 1) verwendet.
Die maximale Größe des Tastaturpuffers beträgt im Normalfall 10. Dies sollte auch als ausreichend angesehen werden. Notfalls können Befehlsworte auch in der abgekürzten Form eingegeben werden. Beim C128 besteht jedoch die Möglichkeit, die Größe des Tastaturpuffers bis auf 20 Elemente zu vergrößern:

```
POKE 2592,20
```

Dabei wird allerdings der TAB-Bitmap-Speicher überschrieben, was zur Folge hat, daß die TAB-Taste nicht mehr sinnvoll benutzt werden kann. Wen das nicht stört, der findet dann einen vergrößerten Tastaturpuffer von:

842 - 861 (vergrößerter Tastaturpuffer beim C128)

vor.

6. DER DATENREKORDER

6. Der Datenrekorder

****

Wir wollen Ihnen in diesem Kapitel ein paar interessante Tricks rund um den Datenrekorder - die Datasette - vermitteln. Sicherlich werden einige hier einwenden, wer sich einen Commodore 128 leiste, werde bestimmt auch nicht auf ein Floppy Disk Laufwerk verzichten.
Wir glauben aber, daß durchaus noch ein großer Anwenderkreis bei der Datasette zu finden ist, da es sich hierbei doch um eine preiswerte Alternative zur Floppy handelt. Außerdem läßt sich die Datasette natürlich auch parallel zur Floppy benutzen. Schließlich gibt es noch Anwendungen, die nur mit dem Datenrekorder funktionieren. Lassen Sie sich überraschen!

6.1. Software-Steuerung der Datasette

Wer gewohnheitsmäßig mit der Datasette zu tun hat, wird die arbeitserleichternde Tatsache gar nicht mehr bemerken: Nach Beendigung eines Lade- oder Speichervorganges stoppt der Motor der Datasette selbsttätig.
Dieser Motorstopp wird durch das Betriebssystem gesteuert, ist also softwaremäßig (aus dem Programm heraus) möglich. Was das Betriebssystem kann, können Sie schon lange und so wollen wir uns die softwaremäßige Motorsteuerung selbst zunutze machen. Wichtig sind in diesem Zusammenhang die Adressen 1 und 192:

```
001  Prozessor-Port
192  Kassetten-Motor-Flag
```

Ohne weiteren Details vorgreifen zu wollen zunächst eine kleine Demo:

Drücken Sie bitte die PLAY-Taste Ihres Rekorders! Der Motor läuft, das Band wird gespult. Jetzt geben Sie bitte die folgende Zeile im Direktmodus ein:

```
POKE 192,1: POKE 1,PEEK(1) OR 32
```

Der Motor der Datasette stoppt, ohne daß eine Taste des Rekorders betätigt wurde. Um den Motor gleichermaßen softwaregesteuert wieder zu starten, genügt:

```
POKE 1,PEEK(1) AND 39: POKE 192,0
```

Sofort arbeitet der Rekorder wieder wie vor dem Eingriff.

Schlüssel ist hier der Prozessor-Port, Adresse 1, auf die wir nun näher eingehen werden. Es interessieren nur die Bits 3 bis 5 dieser Adresse:

```
Bit 3  :  CASS WRT   (Schreiben auf Kassette)
Bit 4  :  CASS SENS  (Abfrage des Rekorders)
Bit 5  :  CASS MOTOR (Motorsteuerung Kassette)
```

Um den Motor der Datasette einzuschalten, müssen das Bit CASS MOTOR gelöscht sowie das Kasettenmotor-Flag $C0 gesetzt werden.
Durch die Software-Steuerung läßt sich beispielsweise zeitgesteuert vor- oder zurückspulen (die entsprechende Taste am Rekorder muß der Anwender jedoch noch selbst drücken!).

6.2. Abfrage der Bandtasten

Aber Adresse 1 kann noch mehr! Interessant ist beispielsweise das Bit 4 dieser Adresse. Hierbei handelt es sich (im Gegensatz zu den anderen beiden Bits) um einen Eingang. Über dieses Bit erfährt der Rechner, ob am Datenrekorder eine Taste gedrückt ist. Dies wird regulär vom Betriebssystem vor jeder Kassettenoperation getan. Ist keine Taste gedrückt, erscheint die Meldung "PRESS PLAY (& RECORD) ON TAPE".
Leider ist es dem Rechner unmöglich, festzustellen, welche der Tasten PLAY, RECORD, REWIND und FFWD gedrückt ist. Dies

werden Anwender vielleicht schon einmal schmerzlich zu spüren bekommen haben, wenn sie versehentlich nach einem Save-Befehl nur die PLAY-Taste gedrückt hatten; der Computer bemerkt diesen Fehler nicht.

Wir können über dieses Bit ebenfalls abfragen, ob eine Taste gedrückt ist. Dies läßt sich zum Beispiel so machen:

```
WAIT 1,16      (wartet, bis die STOP-Taste am Rekorder
                betätigt wird)
WAIT 1,32,32   (wartet, bis eine Taste am Rekorder ge-
                drückt worden ist)

IF (PEEK(1)AND32)=0 THEN PRINT"TASTE GEDRUECKT"
IF (PEEK(1)AND32)=32THEN PRINT"KEINE TASTE GEDRUECKT"
```

Außerdem gibt diese Adresse Aufschluß, ob die Tätigkeit des Motors "gewaltsam" durch Drücken der RUN-STOP-Taste unterbrochen wurde:

```
IF PEEK(1)=99 THEN PRINT"FIES!"
```

6.3. Ungewöhnliches mit dem Datenrichtungsregister

Was ist denn nun schon wieder ein "Datenrichtungsregister", und vor allem: Was kann man damit Ungewöhnliches machen?
Eine ganze Menge, glauben wir. Im Prinzip kann man sich den schon behandelten Prozessor-Port als einen I/O-Port, wie zum Beispiel auch in den CIAs enthalten, vorstellen. Der Prozessor-Port besitzt allerdings nur sechs I/O-Leitungen, wovon drei für Kassettenoperationen benutzt werden.
Diese drei I/O-Leitungen können nun sowohl Eingang als auch Ausgang darstellen. Das heißt, irgendwo muß festgelegt sein, welche Leitung als Eingang und welche als Ausgang definiert ist.
Diese Aufgabe übernimmt das gleichnamige Datenrichtungsregister für den Prozessor-Port, beherbergt in Adresse 0. Die Belegung:

```
Bit 3 : Datenrichtung CASS WRT
Bit 4 : Datenrichtung CASS SENS
Bit 5 : Datenrichtung CASS MOTOR

0 = Eingang
1 = Ausgang
```

Sowohl CASS WRT als auch CASS MOTOR sind Ausgänge. Das ist klar, denn CASS WRT schreibt auf Band und CASS MOTOR steuert den Motor; es wird der Rekorder beeinflußt. Anders CASS SENS. Diese Leitung ist als Eingang geschaltet, denn hierbei wird ja der Rechner beeinflußt: Ist eine Taste gedrückt oder nicht?
Tja, was nun anfangen mit soviel grauer Theorie?

6.4. Ein etwas anderer Kopierschutz

Was hindert uns daran, die Datenrichtung der drei wichtigen Bits ein wenig durcheinander zu bringen? Die folgende Zeile schaltet CASS SENS auf Ausgang:

```
POKE 0,PEEK(0) OR 2^4
```

Der Erfolg ist sicher: Der Rechner kann nun keine Informationen mehr aus CASS SENS beziehen und somit auch nicht mehr registrieren, ob eine Taste am Rekorder gedrückt wurde oder nicht. Die Folge: Nach "LOAD" oder "SAVE" erscheint wie gewöhnlich "PRESS PLAY (&RECORD) ON TAPE". Wird jetzt aber wie gewöhnlich eine Taste gedrückt, passiert gar nichts. Für den Computer gibt es keine Tasten mehr am Rekorder; SAVE und LOAD sind schachmatt.
Zurück gehts wieder, indem man auf Eingang zurückschaltet:

```
POKE 0,PEEK(0) AND NOT 2^4
```

Es gibt aber noch eine interessante Möglichkeit, Programme vor dem Kopieren auf Datasette zu schützen. Man nimmt sich dabei der Leitung CASS WRT an. Über diese Leitung gelangen

normalerweise die Bits aufs Band. Es handelt sich um einen Ausgang. Dieser wird nun einfach als Eingang geschaltet:

```
POKE 0,PEEK(0) AND NOT 2^3
```

Der Befehl "SAVE" scheint ordnungsgemäß zu funktionieren; der Tastendruck wird erkannt und das Band wird (scheinbar) beschrieben. Tatsächlich aber bleiben die Bits im Eingang stecken und gelangen gar nicht erst aufs Band. Der Anwender merkt dies spätestens beim nächsten Laden seines kopierten Programms. Es wird nämlich nichts gefunden (weil nichts auf Band ist...). Der Kopierer wird dann höchstens an der richtigen Funktionsweise seiner Datasette zweifeln und nach einem Kopierschutz gar nicht erst suchen.

Zurück gehts entsprechend:

```
POKE 0,PEEK(0) OR 2^3
```

Als letzte Möglichkeit bietet sich an, die Steuerung des Motors außer Gefecht zu setzen. Wieder wird ein Eingang vorgegaukelt:

```
POKE 0,PEEK(0) AND NOT 2^4
```

Auch hier wieder die Auflösung des Rätsels:

```
POKE 0,PEEK(0) OR 2^4
```

6.5. "HiFi"-Klänge - die Datasette als begnadete Musikbox

Einige Computer besitzen die Möglichkeit, Töne und sogar ganze Musikstücke von einem Datenrekorder abzuspielen. Dies ist bei Commodore-Computern normalerweise nicht möglich. Daß es doch "irgendwie" geht, wollen wir Ihnen auf den nächsten Seiten beweisen. Wir warnen aber vorsorglich: Die Sache hat natürlich auch einen kleinen Haken, wie das nunmal so ist. Die Musik-Wiedergabe-Qualität ist im wahrsten Sinne des Wortes beRAUSCHend. Aber lassen Sie sich überraschen!

In den vorangegangenen Programmen wurden die drei I/O-Leitungen des Prozessor-Ports manipuliert. Dies ist schön und gut, es gibt jedoch keinen Grund, es dabei zu belassen. Vielmehr gibt es eine weitere, bisher unberücksichtigt gebliebene Leitung, die vom Rekorder ins Rechnerinnere führt. Sie hört auf den schönen Namen "CASS READ" und ist das Äquivalent zu "CASS WRT". Es ist ein Eingang, und zwar ein sehr bedeutsamer. Alle vom Rekorder kommenden Daten (z.B. LOAD) laufen über diese Leitung.
Das Problem: Die Leitung endet nicht am Prozessor, wie das bisher mit den anderen Leitungen der Fall gewesen war. Vielmehr strebt sie flugs zum CIA 1 (und steht außerdem noch am seriellen Port zur Verfügung).

Wir werden uns also mit dem Register 13 dieses CIAs näher auseinanderzusetzen haben:

REG 13: Bit 4 : 1=Signal am Pin FLAG aufgetreten

Bevor wir uns jetzt ins Eingemachte stürzen, das Programm:

```
1A00 A0 00     LDY #$00        : Lautstärke Null
1A02 AD 0D DC  LDA $DC0D       : Pin FLAG der CIA 1 abfragen
1A05 C9 00     CMP #$00        : kein Signal ?
1A07 F0 02     BEQ $1A0B       : jawohl, weiter
1A09 A0 0F     LDY #$0F        : nein, Lautstärke aufdrehen
1A0B 8C 18 D4  STY $0418       : und in Register 24 des SID
1A0E A5 D5     LDA $05         : Tastaturabfrage
1A10 C9 58     CMP #$58        : 88=keine Taste gedrückt
1A12 F0 EC     BEQ $1A00       : kein Tastendruck, nochmal
1A14 60        RTS             : zurück zu BASIC
```

Und natürlich wieder ein BASIC-Loader:

(Programm 6.5.a)

Das Prinzip des Programms ist recht einfach (daher die Kürze!): Es wird geprüft, ob am Pin FLAG ein Signal anliegt. Ist das der Fall, wird der Lautsprecher ein-, ansonsten ausgeschaltet. Damit dabei hörbare Töne rauskommen, muß dieses Ein- und Ausschalten extrem schnell erfolgen - und funktioniert deshalb nur in Maschinensprache und in einer möglichst kurzen Schleife.

Noch eine Kleinigkeit zur Tonqualität: Wie bereits angekündigt ist sie außerordentlich verrauscht. Wenn Sie sich mit dieser Routine auf der Datasette eine Musikkassette anhören, werden Sie vielleicht feststellen, woran es mangelt: Die Signale sind stark gefiltert. Sehr hohe und sehr tiefe Töne fallen u.U. weg. Wenn Sie sich Programmkassetten anhören, werden Sie feststellen, daß hierbei die Tonqualität am besten ist; die Filter sind für diese Signale optimiert.
Dennoch lassen sich mit dieser Routine tolle Effekte erzielen. Passen Sie Ihre Stimmlage doch mal der optimalen Tonhöhe an und lassen Sie die "Geisterstimme" in Ihren Programmen ertönen (siehe dazu auch "Softwaresteuerung für Datasette"). Da läßt sich schon eine ganze Menge machen.

6.6. Saven auf Datasette - einmal anders!

Sicher ist Ihnen der SAVE-Befehl nicht unbekannt. Einmal "SAVE" - und sofort (was man nicht unbedingt wörtlich nehmen sollte) ist das BASIC-Programm auf einer Kassette gespeichert.
Sicherlich wissen Sie auch, wie die Programme auf Kassette gespeichert werden: in Form von verschieden langen und hohen Tönen. Das können Sie leicht selbst überprüfen, indem Sie entweder eine Datenkassette in einen normalen Rekorder stecken oder die vorangegangene Routine benutzen.

Wie kommen aber diese Pieps-Töne auf das Band? Wir wollen jetzt nicht die gesamte SAVE-Routine des Betriebssystems auseinanderpflücken. Es geht auch viel einfacher. Wichtig ist im Grunde nur die CASS WRT-Leitung des Prozessor-Ports.

Dieses Bit hat folgende Funktion:

```
Bit 3: CASS WRT
       1=Impuls
```

Es ist also sehr einfach, auf das Band Impulse zu schreiben. Töne entstehen daraus, wenn mehrere Impulse rasch aufeinander folgen. Wir werden das testen:

```
 10 REM TOENE AUF DATASETTE
 20 PRINT "PRESS PLAY & RECORD ON TAPE"
 30 WAIT 1,32,32
 40 FOR W=1 TO 20
 50 FOR K=0 TO 20
 60 POKE 1,PEEK(1)OR8:REM IMPULS
 70 FOR T=1 TO W:REM VERZOEGERUNG
 80 NEXT T
 90 POKE 1,PEEK(1)AND NOT 8:REM ZURUECKSETZEN
100 NEXT K
110 NEXT W
120 END
```

Dieses kleine Programm saved auf Ihre Kassette verschieden hohe und lange Töne. Es soll lediglich das Prinzip veranschaulichen. Da es in BASIC geschrieben ist, sind den Frequenzen der Töne auf Band Grenzen gesetzt, denn die Impulse können nicht beliebig schnell hintereinander auf Band gesaved werden.

7. RUND UM DIE TASTATUR

7. Rund um die Tastatur

****

7.1. Die Tastaturbelegung

Es gibt noch mehr Methoden, die Tastatur abzufragen, als über GET und INPUT. Besonders für Maschinensprache-Programmierer dürften die nun folgenden Methoden interessant sein. Wir wollen dabei zuerst auf die Adresse 213 näher eingehen. In dieser Adresse wird der Wert der augenblicklich gedrückten Taste gespeichert. Dieser Wert stimmt nicht mit den Werten der einzelnen Tasten in der ASCII-Tabelle überein. Sie richten sich vielmehr nach der internen Tastaturbelegung des Rechners.
Die entsprechende Adresse beim C-64 liegt ebenfalls in der Zeropage. Dort wird der Wert der gedrückten Taste in Adresse 203 ($CB) gespeichert. Leider stimmen die Werte beder Rechner nicht immer überein. Daher erst einmal eine Übersicht über die identischen Werte:

Taste	Wert	Taste	Wert
Pfeil links	57	1	56
2	59	3	8
4	11	5	16
6	19	7	24
8	27	9	32
0	35	+	40
-	43	Ö	48
CLR HOME	51	INST DEL	0
CTRL	58	Q	62
W	9	E	14
R	17	T	22
Y	25	U	30
I	33	O	38
P	41	Klammeraffe	46
*	49	^	54
RUN-STOP	63	A	10
S	13	D	18

F	21	G	26
H	29	J	34
K	37	L	42
:	45	;	50
=	53	RETURN	1
Z	12	X	23
C	20	V	31
B	28	N	39
M	36	,	47
.	44	/	55
CRSR down	7	CRSR right	2
SPACE	60		

Wie gesagt, gibt es aber auch Unterschiede. Diese wollen wir hier kurz aufführen :

Keine gedrückte Taste ergibt den Wert 88.
(Beim C-64 ist 64 der entsprechende Wert).

Die folgenden Tasten sind ganz neu:

64	Help-Taste
67	TAB-Taste
72	ESC-Taste
75	LINE FEED

Nun noch eine Besonderheit. Zusätzlich zu den Cursortasten unterhalb der Return-Taste, die die alten Werte behalten haben, gibt es ja noch vier neue gesonderte Cursortasten. Diese haben andere Werte !!!!!

83	Cursor aufwärts
84	Cursor abwärts
85	Cursor links
86	Cursor rechts

Genau wie bei den gesonderten Cursortasten, stimmen die Werte der Tasten des Zehnerblocks nicht mit den Werten der entsprechenden Tasten der alphanumerischen Tastatur überein. Hier sind die Werte für beide Tastaturen :

Taste	Tastatur	Zehnerblock
1	56	71
2	59	68
3	8	79
4	11	69
5	16	66
6	19	77
7	24	70
8	27	65
9	32	78
0	35	81
.	44	82
RETURN	1	76
+	40	73
-	43	74

Außer den Tasten gibt es aber noch ein Gerät, das diese Adresse beeinflußt. Es ist der gute alte Joystick. Allerdings gilt das nur für den Joystickport 1. Doch ist diese Adresse nicht zur Joystick-Abfrage für Spiele geeignet. Drückt man den Joystick nach oben, wird die Adresse nicht verändert. Trotzdem ist dieses Phänomen erwähnenswert. Man könnte nämlich an Port 1 dank dieser Erscheinung noch eine weitere Tastatur mit einigen Tasten anschließen.

```
Joystik oben            88 ;  aber Wirkung wie ALT
Joystick in der Mitte   88 ;  keine Wirkung
Knopf gedrückt          92 ;  Wirkung wie Shift
Joystick links          90 ;  Wirkung wie CTRL
Joystick rechts         91
Joystick unten          89
```

Jetzt haben wir so viel über diese Adresse geredet, daß es endlich Zeit wird, ein Beispiel für den Nutzen dieser Adresse zu liefern.

Mit folgendem Programm kann man den neuen BASIC-Befehl GETKEY simulieren:

```
10 IF PEEK(213) = 88 THEN 10
20 GET A$
30 PRINT A$
40 GOTO 10
```

Natürlich ist diese Befehlsfolge in BASIC-Programmen nicht gerade sinnvoll, da hier der GET KEY-Befehl schon existiert. Gedacht ist dieses Unterprogramm auch eher für Maschinensprache. In Assembler wäre das Programm sehr kurz und könnte etwa so aussehen:

```
loop LDA $D5
     CMP $58
     BEQ loop
```

Nach Verlassen der Schleife enthält der Akku zur Weiterverarbeitung den Wert der gedrückten Taste.

7.2. Ändern der Tastaturbelegung

Jeder Programmierer und jeder Benutzer von Computersystemen hat natürlich individuelle Vorstellungen, wie er den Computer nutzen will. Durch Spiele wurde bei sehr vielen Personen das Interesse am Computer geweckt. Für diese Gruppe wurde in Sachen Bedienungsfreundlichkeit schon an alles, was das Herz begehrt, gedacht. Man kann zwei Joysticks, Lightpens, Trackballs etc. am Computer anschließen.
Eine andere sehr große Gruppe der Computerfans besteht aus Leuten, die über den Beruf zum Computer geführt wurden. Beim C-128 dürfte diese Gruppe u. a. durch CP/M etwas größer als beim C-64 werden.
Um dem Gerät noch etwas weiter auf die Sprünge zu helfen,

ist es oft sinnvoll, die Belegung der Tasten etwas zu verändern.

Dies ist beim C-128 nicht so einfach wie beim C-64. Die Tastatur-Dekodier-Tabelle liegt an zwei Stellen im Rom:

```
64128 ($FA80)   für ASCII angeschaltet
64809 ($FD29)   für DIN angeschaltet
```

Hier kann man die Belegung zwar abfragen (z.B.im Monitor durch den Befehl D FFA80), aber sie läßt sich nicht ändern (wie im ROM allgemein üblich). Allerdings gibt es noch einen anderen Weg. In der Zeropage gibt es einen Zeiger auf die Tastatur-Dekodier-Tabelle. Er liegt bei 830 ($033E) und 831 ($033F). Doch um diese Werte zu ändern, bedarf es eines kleinen Tricks. Es läßt sich normalerweise zwar das Low-Byte ändern, was grundsätzlich dazu führt, daß Sie keine vernünftige Eingabe mehr tätigen können. Aber sobald Sie nur das High-Byte (bei 831) oder den gesamten Zeiger ändern, müssen Sie feststellen, daß der Computer diese Order einfach zu ignorieren scheint. Dies kann aber nicht daran liegen, daß diese Adresse im ROM liegt. Es liegt vielmehr daran, daß im Interrupt laufend die ASCII/DIN-Taste abgefragt wird. Stimmt das High-Byte nicht mit dem für diesen Modus vorgesehenen Wert überein, so wird die Adresse neu gesetzt.
Aber man kann glücklicherweise verhindern, daß die Taste abgefragt wird. Dafür muß das siebte Bit der Adresse 2757 ($0AC5) gesetzt werden:

```
POKE 2757, PEEK(2757) OR 128
```

Nun können Sie den Zeiger auf die Dekodier-Tabelle in aller Ruhe verbiegen, ohne daß er sofort wieder zurückgesetzt wird. An der Stelle, die Sie angeben, sollte sich jedoch wirklich eine sinnvolle Tabelle befinden! Ist dies nicht der Fall, so können Sie häufig keine vernünftige Eingabe mehr machen. Beim Drücken einer Taste erscheint zwar meistens ein Zeichen, der Computer hat sich also nicht aufgehängt, aber es ist selten das gewünschte. Bevor Sie dann einen vernünftigen Text zustande gebracht haben (was oft nicht

funktioniert, weil gar nicht alle Zeichen vertreten sind) ist es praktischer, den Computer auszuschalten, da gegen das Umstellen nicht einmal RUN-STOP/RESTORE hilft.
Kopieren Sie aus diesem Grund vor dem Umschalten einen der Zeichensätze an die von Ihnen gewünschte Stelle, zum Beispiel nach 6912 ($1B00):

```
10 REM Kopieren und Umschalten
20 FOR I=0 TO 88
30 POKE 6912+I, PEEK(64128+I)
40 NEXT I
50 POKE 2757,PEEK(2757) OR 128
60 POKE 830,0
70 POKE 831,27
```

Sollten Sie 'DIN' angeschaltet haben, so werden Sie keinen großen Unterschied feststellen. Haben Sie das Programm dagegen gestartet, als der ASCII-Zeichensatz aktiviert war, haben Sie die Funktion CAPS-LOCK, die bei dem amerikanischen Zwilling ausschließlich angeschaltet ist. Die Funktion CAPS-LOCK hat eine ganz besondere Aufgabe: Durch Drücken dieser Taste erscheinen alle Buchstaben als große Zeichen, genauso, als würden Sie SHIFT oder SHIFT-LOCK betätigen. Der Clou an dieser Taste ist, daß die Zahlen in alter Frische erscheinen und nicht die durch SHIFT erreichbare Doppelbelegung.
Ist CAPS-LOCK nicht gedrückt, so haben Sie die normale Schrift
Zurück zum Ändern der Belegung. Im vorigen Kapitel befindet sich eine Tabelle über die Tastaturbelegung. Diese können Sie jetzt nutzen. Nehmen Sie den Wert der zu tauschenden Taste und addieren Sie ihn zu 6912. In das Ergebnis speichern Sie den ASCII-Wert der neuen Belegung.
Beispiel: Sie wollen Y und Z tauschen. Hängen Sie diese Zeilen einfach an das vorige Programm an:

```
80 POKE 6912+12, 89
90 POKE 6912+25, 90
```

Nach dem Start versteckt sich unter der Z-Taste ein Y und umgekehrt. So einfach geht das.

Jetzt möchten wir Ihnen noch einen Tip geben, wie Sie Maschinensprache-Listings in Form von DATA-Zeilen schneller in den Computer eintippen können. Im Zehner-Block gibt es eine Punkt-Taste. In der Compurtwelt vertritt ja der Punkt bekanntlich das Komma. Dieses Zeichen hat normalerweise nichts mit der Eingabe von ganzzahligen Daten zu tun, wie sie für Maschinensprache-Programme üblich sind. Anstelle eines Punktes wäre also in diesem Fall ein echtes Komma wesentlich sinnvoller.
Also legen wir im Zehner-Block die Punkt-Taste um; ein Komma statt des Punktes:

```
10 REM Komma statt Punkt
20 FOR I=0 TO 88
30 POKE 6912+I,PEEK(64128+I)
40 NEXT I
50 POKE 2757,PEEK(2757) OR 128
60 POKE 830,0
70 POKE 831,27
80 POKE 6994,44
```

7.3. Hex-Tastatur für den C-128

Wenn Sie viel mit hexadezimalen Zahlen arbeiten, kennen Sie sicherlich die Probleme mit der ungünstig angelegten Tastatur. Die Zahlen kann man gut über die Zehnertastatur erreichen, aber die Buchstaben müssen Sie erst zwischen den 61 Tasten der großen Tastatur heraussuchen.
Aber auch dem kann man mit einem kurzen Programm abhelfen. Oberhalb der Zehnertastatur befinden sich die acht Funktionstasten. Diese braucht man nur mit den sechs erforderlichen Buchstaben zu belegen. Nach der Änderung hat man noch zwei Funktionstasten in unmittelbarer Nähe frei. Diese ändern wir gleich mit. Und zwar legen wir auf diese Tasten die Funktionen DEC (" und HEX$(, die als wichtige Umrechnungsbefehle sehr oft gebraucht werden.
Beim Programmieren kommt man häufig in die Verlegenheit, nur

eine Hand frei zu haben, weil die andere mit Büchern etc. beladen ist. Deshalb ist es eigentlich schlecht, wenn zwei Buchstaben nur über die SHIFT-Taste zu erreichen sind. Nun hat man aber noch eine Plus- und eine Minus-Taste, auf die wir bei der Eingabe von Hex-Zahlen ruhig verzichten können. Sie werden also kurzerhand mit den Buchstaben "E" und "F" belegt.

So können Sie "einhändig" eintippen:

```
10 REM Hextastatur auf dem C-128
20 KEY 1,"A"
30 KEY 3,"B"
40 KEY 5,"C"
50 KEY 7,"D"
60 KEY 2,"E"
70 KEY 4,"F"
80 KEY 6,"PRINT DEC("+CHR$(34)
90 KEY 8,"PRINT HEX$("
100 FOR I=0 TO 88
110 POKE 6192+I,PEEK(64128+I)
120 NEXT I
130 POKE 2757, PEEK(2757) OR 128
140 POKE 6994, 32
150 POKE 6985, 69
160 POKE 6986, 70
```

Die Buchstaben "E" und "F" erreichen Sie durch gleichzeitiges Drücken der SHIFT-Taste und F-1 bzw. F-3 oder über die Plus-Taste (E) und die Minus-Taste (F).
Wollen Sie eine hexadezimale Zahl in das dezimale Zahlensystem umrechnen, brauchen Sie nur SHIFT und F-5 zu drücken. Den umgekehrten Weg erreichen Sie durch Shift und F7.

7.4. Doppelbelegungen

Wenn Sie das weiter oben abgedruckte Programm, das Y und Z

vertauscht, ausprobiert haben, ist Ihnen vielleicht aufgefallen, daß die Bedeutungen der mit Shift erreichbaren Funktionen dieser Tasten gleich geblieben sind. Für Doppelbelegungen, die Sie über die SHIFT-, die COMMODORE- und die CTRL-Taste erreichen, gibt es jeweils eine eigene Tabelle.
Zusätzlich existiert noch eine weitere, zur Zeit nicht benutzte Tabelle. Man erreicht sie über ALT. Für ASCII sind diese Adressen geltend:

```
64128 ($FA80)     Erstbelegung
64217 ($FAD9)     mit SHIFT
64306 ($FB32)     mit COMODORE-Taste
64395 ($FB8B)     mit CTRL
64128 ($FA80)     mit ALT (gleiche Bedeutung wie ohne
                  ohne Doppelbelegung)
```

Der Computer hat noch einen zweiten Zeichensatz. Bei gedrückter DIN-Taste wird unter anderem die Tastaturbelegung geändert. Hierfür gibt es wieder 4 vollkommen verschiedene Tabellen, in denen die Tastatur-Dekodier-Tabellen samt den Doppelbelegungen untergebracht wurden. Hier gelten folgende Adressen:

```
64809 ($FD29)     Erstbelegung
64898 ($FD81)     mit gedrückter SHIFT-TASTE
64987 ($FDDB)     mit COMMODORE-Taste
65076 ($FE34)     mit CTRL
64128 ($FA80)     mit ALT
```

Für diese fünf Hilfstasten gibt es in der Zeropage fünf Zeiger auf die jeweilige Tabelle. Außerdem existiert noch ein Zeiger für die ASCII/DIN-Taste:

```
830-831 ($033E-$033F)  ohne Hilfstasten
832-833 ($0340-$0341)  mit SHIFT
834-835 ($0342-$0343)  mit Commodore.
836-837 ($0344-$0345)  mit CTRL
838-839 ($0346-$0347)  mit ALT
840-841 ($0348-$0349)  mit ASCII-Din Taste
```

Die einzelnen Tabellen haben eine Größe von 89 Bytes. Das entspricht den 88 verschiedenen Tasten zusätzlich der Möglichkeit, daß überhaupt keine Taste gedrückt wird.

Sollen zwei Tasten vollständig ausgetauscht werden, so müssen die Werte in allen vier Tabellen entsprechend verändert werden. Das folgende Programm erfüllt diese Aufgabe:

```
10 REM Z und Y vertauschen
15 BANK 15
20 FOR I=0 TO 90*4
30 POKE 6656*I,PEEK(64128+I)
40 NEXT I
50 POKE 2757,PEEK(2757) OR 128
60 REM normale Tabelle
70 POKE 6656+12,89
80 POKE 6656+25,90
90 REM Shift-Tabelle
100 POKE 6656+89+12,89+32
110 POKE 6656+89+25,90+32
120 REM Commodore-Tabelle
130 POKE 6656+2*89+12,183
140 POKE 6656+2*89+25,184
150 REM CTRL
160 POKE 6656+3*89+12,25
170 POKE 6656+3*89+25,26
180 REM Zeiger verbiegen
190 POKE 830,0
200 POKE 831,26
210 POKE 832,89
220 POKE 833,26
230 POKE 834,89*2
240 POKE 835,26
250 POKE 836,11
260 POKE 837,27
```

Jetzt können Sie auch auf die Doppelbelegungen zurückgreifen

und diese nach Belieben ändern. Die neue Tastaturbelegung brauchen Sie nicht unbedingt an die Adresse 6656 zu legen, wenn dieser Speicherplatz anderweitig belegt ist. Denken Sie daran, daß Sie gleichzeitig auch die Zeiger (830-841) ändern müssen.

7.5. Die Hilfstasten

Wie oben schon erwähnt, braucht der Computer einige Hilfstasten, um die vielen Doppelbelegungen zu realisieren. Beim C-64 gibt es drei Hilfstasten: SHIFT (beide SHIFT-Tasten und Shift-Lock haben genau die gleiche Bedeutung),COMMODORE und CTRL.
Beim C-128 sind noch drei weiter Hilfstasten dazu gekommen: ESC, ALT und die Umschalttaste ASCII/DIN.
ESC steht für Escape (Flucht). Diese Taste ist keine Neuerfindung von Commodore. Es gibt Sie schon seit Jahren bei vielen Computern. Mit der ESC-Taste erreichen Sie auf dem C-128 mehrere Funktionen, wie das Umschalten vom 40- auf den 80-Zeichen Bildschirm. Jede Taste hat in Verbindung mit der ESC-Taste eine spezielle Bedeutung. ESC unterscheidet sich aber von den anderen SHIFT-, COMMODORE und CTRL-Tasten: ESC und eine andere Taste müssen nicht gleichzeitig sondern nacheinander gedrückt werden.
Den Zweck der ASCII/DIN-Taste haben wir schon in einem der oberen Kapitel erwähnt. Sie dient zum Umschalten vom ASCII- oder DIN-Zeichensatz, kann aber auch zu CAPS-Lock umfunktioniert werden, was bestimmt auch seine Reize hat.
Die Funktion der ALT-Taste ist schwerer zu ersehen. Eine Abfrage der Adresse 213 ergibt auch hier den für die Hilfstasten üblichen Wert Null. Das heißt aber nicht, dieser Taste wäre in der Tastatur-Decodier-Tabelle kein Platz gewidmet. Auch wenn man diese Taste wie die SHIFT-Taste oder nach dem Vorbild der Escape-Taste benutzt, passiert herzlich wenig. Es ist nämlich noch keine Funktion für diese Taste vorgesehen.

7.5.1. Nutzung der Hilfstasten

Die Hilfstasten werden grundsätzlich über die Adresse 211 ($D3) abgefragt.
Ist eine der SHIFT-Tasten oder SHIFT-LOCK gedrückt, so ist das erste Bit gesetzt. Bei der COMMODORE-Taste ist das zweite Bit gesetzt. Das dritte Bit ist für CTRL und das vierte für ALT. Das nächste Bit ist gesetzt, wenn die ASCII/DIN-Taste gedrückt ist.

Da diese Tasten auch zusammen gedrückt werden können, existieren 31 verschiede Kombinationen. Die folgende Tabelle zeigt ein paar dieser Möglichkeiten. Sie ist unvollständig und soll nur das Prinzip erklären. Wenn Sie das Schema verstanden haben, ist eine vollständige Tabelle nicht notwendig.

0	- Keine Hilfstaste	1	- SHIFT
2	- COMMODORE-TASTE	3	- SHIFT + COMMODORE
4	- CTRL-Taste	5	- SHIFT + CTRL
8	- ALT	10	- ALT + Commodore
15	- SHIFT+COMMODORE+CTRL+ALT	16	- ASCII/DIN-Taste

7.6. Vier zusätzliche Funktionstasten

Wenn Ihnen die acht schon vorhandenen Funktionstasten nicht ausreichen- zum Beispiel weil Sie mit der HEX-Tastatur arbeiten und die normalen Funktionstasten den HEX-Tasten zum Opfer gefallen sind- haben Sie mit diesem Programm die Lösung Ihres Problems.

Auf jede der neuen Funktionstasten können Sie einen sechzehn Zeichen langen Text legen, ähnlich wie bei dem KEY-Befehl. Sie können auch alle Steuerzeichen mit eingeben. Die zusätzlichen Funktionstasten erreichen Sie über die ALT-Taste. Die erste neue Funktionstaste erreichen Sie, wenn Sie nur ALT drücken, die zweite mit ALT und SHIFT. Über COMMODORE und CTRL in Verbindung mit ALT wirken die dritte und die vierte. Als "Abfallprodukt" fallen noch vier weitere Funktionstasten ab, die aber wesentlich schlechter zu erreichen sind. Sie müssen mehrere Hilfstasten und ALT gleichzeitig drücken.
In dem BASIC-Lader ist gleich ein Programm enthalten, mit dem Sie die acht neuen Funktionstasten belegen können.
Sie können die Eingabe nicht auf dem Bildschirm verfolgen. Erst wenn sie abgeschlossen ist, wird der Text ausgegeben und gefragt, ob die Eingabe so richtig war.

(Programm 7.6.a)

Wenn Sie eigene Verwendungen für die ALT-Taste programmieren, können Sie nach dem gleichen Prinzip, das in dem folgenden Maschinensprache-Programm angewandt wurde, vorgehen.

```
1400  78        SEI          ; verhindert Interrupt
1401  A9 0D     LDA #$0D     ; setzt Interruptvektor
1403  8D 14 03  STA $0314
1406  A9 14     LDA #$14
1408  8D 15 03  STA $0315
140B  58        CLI          ; erlaubt IRQ
140C  60        RTS          ; Zurück ins BASIC
140D  48        PHA          ; Akku auf Stack
140E  98        TYA          ; Y-Register in AKKU
140F  48        PHA          ; Akku auf Stack
1410  A5 D3     LDA $D3      ; Flag für ALT.
1412  A8        TAY          ; Akku in Y-Register
1413  29 08     AND #$08     ; Bit 3 gesetzt ?
1415  F0 1D     BEQ $1434    ; nein, dann $1434
1417  98        TYA          ; Wert zurück in Akku
1418  CD 60 14  CMP $1460    ; vorherige Taste ALT?
141B  F0 17     BEQ $1434    ; ja, dann $1434
141D  8D 60 14  STA $1460    ; speichert 8 ab
1420  0A        ASL          ; verschiebt um vier Bits
1421  0A        ASL          ; nach links
1422  0A        ASL
1423  0A        ASL
1424  A8        TAY          ; Akku in Y-Register.
1425  B9 00 14  LDA $1400,Y  ; Zeichen
1428  F0 0D     BEQ $1437    ; $1437 wenn Wert = 0
142A  C9 0D     CMP #$0D     ; Return?
142C  F0 0F     BEQ $143D    ; ja, dann $143D
142E  20 D2 FF  JSR $FFD2    ; Zeichen ausgeben
1431  C8        INY
1432  D0 F1     BNE $1425
1434  8C 60 14  STY $1460    ; Speichert Hilfstaste
1437  68        PLA          ; Akku aus Stack
```

```
1438  A8        TAY          ; Akku ins Y-Register
1439  68        PLA          ; Akku aus Stack
143A  4C 65 FA  JMP $FA65    ; Zum normalen Interrupt
143D  80 4A 03  STA $034A    ; Return in Tast.-puffer
1440  A9 01     LDA #$01
1442  85 D0     STA $00
1444  4c 37 14  JMP $1437
```

7.7. Tastatur-Piep

Nun noch ein kleines Programm, das Ihrem Rechner ein etwas professionelleres Aussehen gibt: Bei jedem Tastendruck wird ein Piep-Ton ausgegeben - wie Sie es vielleicht von größeren Rechnern her kennen.
Damit das Programm auf jeden Tastendruck schnell reagiert, liegt es im IRQ.
Aus diesem Grund ist es, wie Sie sich sicherlich gedacht haben, nicht in BASIC geschrieben, kann aber mit einem BASIC-Lader in den Computer eingegeben werden. Haben Sie das Programm erst einmal gestartet, können Sie unabhängig von diesem Programm ganz normal weiter programmieren. Um die Eingabe noch weiter zu erleichtern, stößt Ihr Rechner einen zufriedenen Brummton bei jedem Druck der RETURN-Taste aus. Wenn Sie schon ein wenig Maschinensprache beherrschen, können Sie den genauen Programmverlauf aus dem nun folgenden Assembler-Listing entnehmen.

```
     1400 A9 FF     LDA #$FF        ; Setzt Werte für
     1402 80 06 D4  STA $D406       ; den Piepton
     1405 8D 18 D4  STA $D418
     1408 A9 09     LDA #$09
     140A 80 05 04  STA $0405
     140D 78        SEI             ; Verhindert Interrupt
     140E A9 1A     LDA #$1A
     1410 8D 14 03  STA $0314       ; IRQ-Vektor auf
     1413 A9 14     LDA #$14
; Piepton-Routine
     1415 8D 15 03  STA $0315       ; setzen
     1418 58        CLI             ; Erlaubt Interrupt
```

```
1419 60        RTS          ; Rückkehr ins BASIC
141A 48        PHA          ; Akku auf Stack
141B A5 D5     LDA $D5      ; Wert der gedrückten
                            ; Taste
141D C9 58     CMP #$58     ; Taste gedrückt?
141F F0 24     BEQ $1443    ; ja
1421 C9 4C     CMP #$4C     ; RETURN?
1423 D0 0D     BNE $1431    ; nein, dann $1431.
1425 A9 67     LDA #$67     ; speichert Frequenzen.
1427 BD 00 D4  STA $D400    ; Ton bei RETURN
142A A9 11     LDA #$11
142C 8D 01 D4  STA $D401
142F D0 14     BNE $1445
1431 C9 01     CMP #$01     ; andere RETURN-Taste?
1433 F0 F0     BEQ $1425    ; ja, dann $1425.
1435 A9 67     LDA #$67     ; setzt Frequenz für
1437 8D 01 D4  STA $0401    ; den Piepton
143A A9 21     LDA #$21
143C 8D 00 D4  STA $D400
143F A9 11     LDA #$11     ; Wellenform
1441 D0 02     BNE $1445    ; nächste Zeile
                              überspringen
1443 A9 00     LDA #$00     ; Wellenform, wenn
                              keine Taste gedrückt
1445 8D 04 D4  STA $D404    ; Wellenform speichern
1448 68        PLA          ; Akku aus Stack
144A 4C 65 FA  JMP $FA65    ; Normaler IRQ
```

Hier für alle die, die das Programm lieber in BASIC eingeben möchten, der BASIC-Lader:

(Programm 7.7.a)

7.8. Die STOP-Taste

Mit dieser Routine wird die Bedeutung der NO-SCROLL-Taste erweitert: Statt nur die Ausgabe auf den Bildschirm anzuhalten, kann der gesamte Programmablauf unterbrochen

werden. Damit kann man ein Spiel ruhigen Gewissens verlassen, um z. B. zum Telefon oder zur Haustüre zu gelangen.

Wenn Sie die Stopfunktion lieber auf eine andere Taste als die NO-SCROLL-Taste gelegt haben wollen, so müssen Sie die folgenden POKE-Befehle an das Ende des Laders hängen:

```
POKE 5134, N
POKE 5140, N
```

Der Parameter N muß den Wert für die von Ihnen gewünschte Taste enthalten. Die genauen Werte erfahren Sie im Kapitel "Tastaturbelegung".

Mit dem unterbrochenen Programm wird fortgefahren, sobald eine beliebige andere Taste gedrückt wird. Diese Taste bleibt im Tastaturpuffer und wird deshalb bei der nächsten Eingabe berücksichtigt.

(Programm 7.8.a)

Sie können die Routine natürlich auch in Maschinensprache eingeben oder von dort starten:

```
1400  A9 0B     LDA #$0B      ; Verändert Vektor
1402  8D 08 03  STA $0308     ; für "nächste BASIC-
1405  A9 14     LDA #$14      ; Zeile ausführen "
1407  8D 09 03  STA $0309
140A  60        RTS           ; Zurück ins BASIC
140B  A5 D4     LDA $D4       ; Wert der Taste
140D  C9 40     CMP #$57      ; NO SCROLL?
140F  D0 0A     BNE $141B     ; nein
1411  A5 D4     LDA $D4       ; Wert der Taste
1413  C9 40     CMP #$57      ; NO SCROLL?
1415  F0 FA     BEQ $1411     ; ja, dann warten
1417  C9 58     CMP #$58      ; Keine Taste?
1419  F0 F6     BEQ $1411     ; ja, dann warten
141B  A9 0F     LDA #$0F      ; Nächste BASIC-Zeile
```

```
141D  85 02    STA $02        ; ausführen
141F  A9 4E    LDA #$4A
1421  85 03    STA $03
1423  A9 A2    LDA #$A2
1425  85 04    STA $04
1427  A9 00    LDA #$00
1429  85 05    STA $05
142B  4C E3 02 JMP $02E3
```

7.9. Belegung der HELP-Taste

Der C-128 hat insgesamt zehn Funktionstasten, die Sie alle selbst belegen können. Außer den üblichen acht Funktionstasten, oberhalb der Zehnertastatur, gibt es noch zwei weitere: eine dieser beiden ist die HELP-Taste, die aber weniger unter dem Namen Funktionstaste 10 bekannt ist. Die neunte Funktionstaste erreicht man durch Drücken von SHIFT und RUN-STOP. Diese Taste ist von vornherein mit

```
DLOAD
RUN
```

belegt. Diese Zeilen sorgen dafür, daß das erste Programm auf der Floppy geladen und sofort gestartet wird.
Aber auch diesen Text kann man ändern, jedoch nicht über den KEY-Befehl. Bei Eingabe von KEY 9 oder KEY 10 meldet sich der Computer mit "?ILLEGAL QUANTITY ERROR".
Ein kleines Programm kann da jedoch weiterhelfen:

(Programm 7.9.a)

Mit diesem Programm können diese beiden Funktionstasten neu belegt werden. Soll nach dem Text ein RETURN ausgeführt werden, so muß ein Pfeil nach oben eingegeben werden. Damit auch Kommas benutzt werden können, haben wir die Eingabe über GET gewählt.
Nach geringen Änderungen können mit dem Programm auch die anderen Funktionstasten belegt werden. Das kann besonders dann nützlich sein, wenn man einen Text auf eine

Funktionstaste legen möchte, der aus mehr als 128 Zeichen besteht. Beim KEY-Befehl würde in diesem Fall eine Fehlermeldung ausgeben werden.

Für die Belegung der Funktionstasten ist der Bereich von 4096 ($1000) bis 4AA2 ($1100) vorgesehen. In den ersten zehn Bytes ist die Länge der einzelnen Aexte abgelegt. Mit Hilfe dieser Werte lassen sich auch die Startadressen der Texte errechnen. Zählt man beispielsweise die ersten 5 Bytes zusammen, erhält man die Anfangsadresse des Textes zu F6. Sie können insgesamt 245 Zeichen in die 10 Funktionstasten legen. Das Programm verhindert selbst, daß ein längerer Text eingegeben wird.
Bei der Eingabe können Sie außer RETURN natürlich noch andere Steuerzeichen benutzen. Diese können direkt eingegeben werden. Wollen Sie zum Beispiel den Bildschirm zehnmal nach oben scrollen, so können Sie einfach zehnmal ESC und W drücken. Wenn Sie ein falsches Zeichen eingegeben haben, sollten Sie nicht die INST DEL-Taste benutzen. Diese zeigt anscheinend auch die richtige Wirkung, aber im Speicher wird die falsche Eingabe und INST DEL abgespeichert und jedesmal mit ausgegeben.

8. BEFEHLSERWEITERUNG

8. Befehlserweiterung - selbst gemacht

****

Der C-128 meldet sich nach dem Einschalten unter anderem mit dem Schriftzug "(C) 1977 MICROSOFT CORP.". Von Microsoft wurde die bekannte und allseits beliebte BASIC-Version MBASIC entworfen, die hauptsächlich auf CP/M-Rechnern eingesetzt wird. Diese BASIC-Version ist für ihren großen Befehlssatz bekannt. Daher ließe sich vermuten, daß die BASIC-Version des C-128 auch über viele Befehle verfügt. Wie Sie sich durch einen Blick ins Handbuch vergewissern können, ist dies auch geschehen. Diese BASIC-Version trägt die Bezeichnung V7.0 und ist aufwärtskompatibel zu den BASIC-Versionen V2.0, V4.0 und V3.5. Die Version V2.0 ist praktisch der Urahn der anderen BASIC-Versionen; auch der VC-20 und der C-64 enthalten dieses BASIC. BASIC V4.0 wurde in den "großen" Commodore-Rechnern implementiert, so z. B. in den 8000er Serien. Zu den Befehlen des BASIC V2.0 sind Befehle zum Umgang mit Diskettenstationen gekommen (z. B. DIRECTORY). Die nächste BASIC-Version, V3.5, wurde auf den Rechnern C16, C116 und Plus/4 installiert. Diese BASIC-Version enthält endlich auch Grafik-Befehle. Auch Programmierhilfen sind dazugekommen (z. B. RENUMBER, AUTO, DELETE). Als Clou wurde noch ein Monitor eingebaut, der mit dem Befehl MONITOR (wie sinnvoll) aufgerufen werden kann.
Nun kommt die Version V7.0. Hier sind noch einige Befehle zur Sprite-Steuerung, Musik-Programmierung (in der Version V3.5 nur beschränkt eingebaut) und zum Diskettenumgang dazugekommen. Auch hier existiert ein "Clou": der Befehl GO 64. Mit ihm kommt man in den C-64 Modus (ob dieser Befehl wohl auch noch bei neueren Rechnern von Commodore dabeisein wird?).
Trotz dieser zahlreichen Befehle kommt es oft vor, daß noch weitere erwünscht werden. Wie wäre es z. B. mit PRINT AT? Oder POP? Oder GOTO X? Und RESTORE X? Die Liste ließe sich beliebig fortsetzen.
Als engagierter Programmierer wird es nur kurze Zeit dauern, und man besitzt zahlreiche dieser Hilfsroutinen. Und dann? Will man sie aufrufen, so geschieht das meistens mit einen

SYS und einigen Werten dahinter. Doch wie viele Adressen kann man sich merken? Und wie verständlich wirkt ein Programm, das nur aus einer Aneinanderreihung von SYS-Befehlen besteht?
Zum Glück besteht bei den Commodore-Rechnern aber eine einfache Möglichkeit, eigene Befehle in das Betriebssystem einzufügen. Dies geschieht mit der sog. CHRGET-Routine (CHRGET steht für character + get).

8.1. Was ist die CHRGET-Routine?

Zuerst einmal: Die CHRGET-Routine wird beim Einschalten des Rechners vom ROM ins RAM kopiert. Zu dem Sinn des ganzen kommen wir gleich noch.
Nun zu der Funktion der CHRGET-Routine:
Mit der CHRGET-Routine wird ein Zeichen aus dem BASIC-Text, d. h. aus dem Programmspeicher oder aus dem BASIC-Eingabe-Puffer, geholt. Diese CHRGET-Routine wird vom BASIC-Interpreter jedesmal angesprungen, wenn das nächste Zeichen geholt werden muß. So müssen z. B. beim SYS-Befehl noch weitere Zeichen geholt werden, nämlich die Startadresse und ggf. die Register.
Schauen wir uns dazu einmal die CHRGET-Routine, erst einmal beim C-64, an:

```
0073   INC $7A
0075   BNE $0079
0077   INC $7B
0079   LDA $HHLL
007C   CMP #$3A
007E   BCS $008A
0080   CMP #$20
0082   BEQ $0073
0084   SEC
0085   SBC #$30
0087   SEC
0088   SBC #$D0
008A   RTS
```

Die ersten drei Befehle erhöhen einen Zeiger. Dabei wird das Low-Byte und bei einem Übertrag, d. h. wenn das Low-Byte größer als 255 wurde, auch das High-Byte. Wenn Sie sich die Adresse dieses Zeigers einmal anschauen, fällt auf, daß er innerhalb der CHRGET-Routine liegt! Jetzt wird auch der Sinn der Kopiererei klar: die Routine verändert sich selbst!
Das geschieht dann schon im nächsten Befehl. Hier ist nämlich der angesprochene Vektor zu finden. Das heißt also, der Akkumulator wird in der Adresse $0079 immmer mit dem Inhalt der Speicherstelle geladen, auf die der Vektor zeigt. Dieser Vektor wird dann einfach auf den BASIC-Text gelegt und liest so ein BASIC-Zeichen.
Dieses Zeichen wird nun verschiedentlich geprüft.
Zuerst wird das eingelesene Zeichen mit dem ASCII-Wert für den Doppelpunkt verglichen. Ist das Zeichen größer oder gleich dem Wert, so wird gleich zurückgesprungen. War das Zeichen gleich dem Doppelpunkt, so wird zusätzlich zu dem Carry-Flag noch das Zero-Flag gesetzt.
War das Zeichen kleiner (so z. B. eine Zahl), so werden noch weitere Tests ausgeführt. Zuerst wird getestet, ob das Zeichen ein Leerzeichen ist. Ist dies der Fall, so wird gleich das nächste Zeichen geholt. Leerzeichen werden also überlesen (PRINTX hat die gleiche Wirkung wie PRINT X).
Die nächsten beiden Befehle testen dann, ob das eingelesene Zeichen eine Ziffer zwischen 0 - 9 darstellt.

Die CHRGET-Routine des C-128 wurde nun etwas verändert:

```
0380   INC $30
0382   BNE $0386
0384   INC $3E
0386   STA $FF01
0389   LDY #$00     038B   LDA ($3D),Y
0380   STA $FF03
0390   CMP #$3A
0392   BCS $039E
0394   CMP #$20
```

```
0396   BEQ $0380
0398   SEC
0399   SBC #$30
039B   SEC
039C   SBC #$D0
039E   RTS
```

Die ersten drei Befehle erhöhen wieder einen Vektor. Dieser liegt jetzt jedoch nicht mehr innerhalb der CHRGET-Routine, sondern in der Zeropage.
Der nächste Befehl wurde eingefügt, damit der Computer sich seine Information immer von der richtigen Bank holt. Normalerweise entspricht dieser Befehl dem BASIC-Befehl BANK 0. Es wird also durchgehend die RAM-Bank 0 eingeschaltet. Dadurch wird gewährleistet, daß das Programm die vollen 64K der RAM-Bank 0 belegen kann (abzüglich der ersten Pages, die vom System benutzt werden). Würde hier nicht umgeschaltet werden, so würde der Computer u. U. auf das ROM zurückgreifen, was natürlich nicht gerade sinnvoll wäre.
Die nächsten beiden Befehle holen das Zeichen aus dem BASIC-Text. Da das Y-Register als Offset benutzt wird, muß es vorher auf Null gesetzt werden (ach ja, warum besitzt der 6502 bloß keine indirekte Adressierung ohne Indizierung?).
Der Befehl in Adresse $038D schaltet die Banks wieder zurück. Dieser Maschinensprache-Befehl entspricht dem BASIC-Befehl BANK 14, d. h. es wird Kernal mit BASIC-Interpreter, Zeichengenerator und RAM-Bank 0 eingeschaltet.
Die Befehle danach entsprechen denen beim C-64, d. h. es wird wieder auf Doppelpunkt, Leerzeichen etc. geprüft.

Noch ein Abschlußwort zu der CHRGET-Routine: Ab und zu wird im Betriebssystem die CHRGET-Routine auch bei Adresse $0386 ($0079 beim C-64) angesprungen. Sie heißt dann CHRGOT. Wie Sie schnell feststellen können, wird auch hier ein Zeichen geholt, allerdings ohne daß der Zeiger vorher inkrementiert wurde. Dadurch wird das letzte Zeichen eingelesen.

Kommen wir nun zum Wichtigsten:

8.2. Wie verändert man die CHRGET-Routine?

Bei den Manipulationen wollen wir uns nur auf den C-128 beziehen, da zum C-64 schon genug Literatur vorhanden ist.
Um eigene Befehle einzubinden, muß man die CHRGET-Routine verändern. Doch wo macht man das? Theoretisch ginge das überall, praktisch sind jedoch nur zwei Fälle interessant:

a) Es soll jedesmal, wenn ein Zeichen geholt wird, eine bestimmte Funktion ausgeführt werden.
b) Es soll ein neuer Befehl eingebunden werden.

Ist a) der Fall, so verändert man die CHRGET-Routine am besten am Anfang. Dazu gleich ein Beispiel. Gehen Sie in den Monitor und tippen Sie das folgende Assembler-Listing ab:

```
0380   JSR $0B00
0383   NOP
0384   NOP
0385   NOP
---------------------
0B00   INC $3D
0B02   BNE $0B06
0B04   INC $3E
0B06   LDA #$00
0B08   STA $FF00
0B0B   LDA $D020
0B0E   EOR #$FF
0B10   STA $D020
0B13   RTS
```

Verlassen Sie nun den Monitor mir "X", tippen Sie ein Zeichen ein und drücken Sie die Return-Taste. Wie Sie sehen, wird die Rahmenfarbe verändert (die Rahmenfarbe des 40-Zeichen-Bildschirms, da auf den VIC zurückgegriffen wird).

Schauen wir uns also einmal das Listing an:
Es besteht aus zwei Teilen. Der erste Teil liegt innerhalb der CHRGET-Routine und verändert diese also. Wird nun ein Zeichen eingegeben, so wird ein Unterprogramm bei $0B00 aufgerufen. Dieses Unterprogramm stellt den zweiten Teil dar. Hier wird zuerst der Teil der CHRGET-Routine weitergeführt, der durch den ersten Teil gelöscht wurde. Dann wird die I/O eingeschaltet (Bit 0 ist dafür zuständig, siehe auch Kapitel 9 "Banking"), damit auf den VIC zugegriffen werden kann. Der Zugriff geschieht mit den nächsten drei Befehlen. Hier wird die Rahmenfarbe invertiert. Dann wird wieder die Bank eingeschaltet, die vor dem Aufruf der Routine gegeben war. Schließlich geht es durch ein RTS in der CHRGET-Routine weiter.
Eine Sache müssen Sie also beachten: Die CHRGET-Routine muß vollständig erhalten bleiben. Ist dies nicht der Fall (lassen Sie z. B. einmal in dem zweiten Assembler-Listing die Befehle weg, die den Zeiger innerhalb der CHRGET-Routine erhöhen), so wird das Betriebssystem nicht mehr richtig funktionieren.

Doch ist es in den wenigsten Fällen sinnvoll, daß eine Aktion bei jedem eingegebenen Zeichen, egal was es ist, ausgeführt wird. Meistens will man eine Aktion nur bei einem bestimmen Zeichen (oder natürlich auch bei einer bestimmten Zeichenfolge) ausführen lassen. Diese Zeichenfolge stellt dann den neuen Befehl da.
Wie Sie sich denken können, kann man auch das mit der CHRGET-Routine erledigen. Geben Sie dazu folgendes Assembler-Listing ein (schalten Sie den Rechner am besten einmal vorher aus):

```
0380   INC $3D
0382   BNE $0386
0384   INC $3E
0386   STA $FF01
0389   LDY #$00
038B   LDA ($3D),Y
038D   JMP $0B00
```

```
0390   STA $FF03
0393   NOP
---------------------
0B00   CMP #$FF
0B02   BNE $0B14
0B04   LDA #$00
0B06   STA $FF00
0B09   LDA $D020
0B0C   EOR #$FF
0B0E   STA $D020
0B11   JMP $0380
0B14   CMP #$3A
0B16   BCS $0B1B
0B18   JMP $0390
0B1B   STA $FF03
0B1E   RTS
```

Diesmal wurde die CHRGET-Routine erst bei Adresse $038D verändert. Dort wird zu Adresse $0B00 gesprungen (diesmal allerdings nicht JSR sondern JMP!). Der Befehl in Adresse $0390 (STA $FF03) wurde wieder von der alten CHRGET-Routine übernommen. Nach dem NOP-Befehl, der das durch die Manipulation freigebliebene Byte innerhalb der CHRGET-Routine belegt, geht es normal weiter.
Nun zu dem Teil ab Adresse $0B00:
Zuerst wird das eingelesene Zeichen mit dem Wert $FF (255) verglichen. Dieser Wert stellt den Token für Pi da. Ist das eingelesene Zeichen nicht Pi, so geht es ab Adresse $0B14 weiter. Dort wird der weggefallene Teil der CHRGET-Routine fortgeführt. Das Zeichen wird zuerst mit dem Doppelpunkt verglichen. Ist es größer, so wird Bank 14 eingestellt (wie dies auch bei der CHRGET-Routine der Fall war) und zu der Adresse zurückgesprungen, von der die CHRGET-Routine aufgerufen worden war. War der ASCII-Wert des Zeichens kleiner als des Doppelpunkts (z. B. eine Ziffer), so geht es durch den JMP-Befehl normal in der CHRGET-Routine weiter.
Doch nehmen wir einmal an, das eingegebene Zeichen war Pi. In diesem Fall wird das Programm bei der Adresse $0B04 fortgeführt. Hier wird zuerst die Bank 15 eingestellt, damit

wieder auf die I/O (in diesem Fall auf den VIC) zugegriffen werden kann. Dann wird, wie in der oberen Routine, der Wert für den Bildschirmrahmen invertiert. Schließlich wird wieder zu der CHRGET-Routine zurückgesprungen, damit das nächste Zeichen gelesen wird.
Geben Sie also einmal Pi ein und drücken Sie die RETURN-Taste. Die Rahmenfarbe des 40-Zeichen-Bildschirms wird verändert. Geben Sie nun noch einmal Pi ein: Die Rahmenfarbe wird wieder normal.
Zügeln Sie nun etwas Ihre Neugier und probieren Sie mit dem Pi-Zeichen nicht weiter rum (z. B. wie es sich verhält, wenn man es innerhalb einer Programmzeile eingibt). Tippen Sie vorher folgendes Listing ab:

```
0380   INC $3D
0382   BNE $0386
0384   INC $3E
0386   STA $FF01
0389   LDY #$00
038B   LDA ($3D),Y
038D   JMP $0B00
0390   STA $FF03
0393   NOP
------------------
0B00   CMP #$FF
0B02   BNE $0B14
0B04   LDA #$00
0B06   STA $FF00
0B09   LDA $0020
0B0C   EOR #$FF
0B0E   STA $D020
0B11   JMP $0380
0B14   CMP #$88
0B16   BNE $0B28
0B18   LDA #$00
0B1A   STA $FF00
0B1D   LDA $D021
0B20   EOR #$FF
0B22   STA $D021
```

```
0B25   JMP $0380
0B28   CMP #$3A
0B2A   BCS $0B2F
0B2C   JMP $0390
0B2F   STA $FF03
0B32   RTS
```

In dieses Listing werden zwei neue Befehle eingefügt:

a) Pi: Hier wird wie bei der vorigen Routine der Wert für die Rahmenfarbe invertiert.
b) LET: Dieser Befehl invertiert den Wert für die Hintergrundfarbe.

Auf das Listing wollen wir nicht weiter eingehen, da es ziemlich leicht zu verstehen ist. Schauen Sie es sich trotzdem noch einmal an.

8.3. Das "Verhalten" der neuen Befehle

Kommen wir nun zu dem "Verhalten" dieser neuen Befehle:
Geben Sie einmal das Zeichen Pi mit einer Zeilennummer davor ein. Wie Sie sehen, wird der Befehl trotzdem sofort ausgeführt und die Zeile wird nicht ins Programm aufgenommen. Nun probieren Sie das gleiche einmal mit dem neuen alten Befehl "LET" aus. Hier wird der Befehl nicht sofort ausgeführt, sondern in die Zeile übernommen und erst bei RUN ausgeführt. Der Computer unterscheidet also zwischen normalen und neuen BASIC-Befehlen. Denken Sie nun aber nicht, daß Sie Ihre neuen Befehle nicht innerhalb von Programmen benutzen können! Geben Sie noch einmal Pi in einer Zeile ein, tippen Sie nun jedoch an den Anfang der Zeile einen Doppelpunkt. Nun wird zwar der Befehl schon wieder sofort ausgeführt, die Zeile bleibt jedoch erhalten. Das geht auch, wenn Sie den neuen Befehl zwischen alten Befehlen einfügen. Setzen Sie also statt des Doppelpunkts den Befehl LET in die Zeile. Auch in diesem Fall bleibt die Zeile erhalten, und wenn Sie sie starten, so werden die Rahmen- und die Hintergrundfarbe verändert. Probieren Sie auch einmal folgende Zeile aus:

```
10 LET GOTO 10
```

Obwohl sie falsch aussieht, funktioniert sie richtig: Der Wert der Hintergrundfarbe wird ziemlich schnell invertiert (der Effekt, der sich hier ergibt, ist recht interessant).
Das funktioniert auch in folgendem Format:

```
10 GOTO LET 10
```

Woran liegt das? Bei der Interpretation der BASIC-Zeilen wird die (veränderte) CHRGET-Routine angesprungen. Hier wird zuerst der Token für GOTO gelesen. Da dieser Wert nicht mit einem der Werte der beiden neuen Befehle übereinstimmt, wird wieder zur Auswertung zurückgesprungen und zum GOTO-Befehl verzweigt. Dort wird die CHRGET-Routine noch einmal angesprungen, um die Zeilennummer zu holen. Diesmal stimmt das Zeichen jedoch mit einem Wert der neuen Befehle überein,

nämlich mit dem Wert $88 für LET. Also wird LET ausgeführt und wieder zur CHRGET-Routine gesprungen. Dort wird das nächste Zeichen, die Ziffer 1, gelesen und zum GOTO-Befehl zurückgesprungen. Der Befehl LET wird also ausgeführt, ansonsten jedoch einfach überlesen. Das funktioniert natürlich auch hinter jedem anderen Befehl. Allerdings gibt es auch einige Einschränkungen:

a) Der neue Befehl darf natürlich nicht innerhalb von anderen Befehlen stehen. Folgendes geht also nicht:

```
10 GO LET TO 10
```

Das liegt daran, daß GOTO erst in einen Token umgewandelt wird. In diesem Fall wird der Befehl GOTO jedoch gar nicht erst erkannt und so auch nicht umgewandelt.
Man kann den neuen Befehl jedoch innerhalb von Zahlen plazieren:

```
10 A=100LETO
20 PRINT A
```

Hier erhält A den Wert 1000.

b) Der neue Befehl wird nicht innerhalb von Strings erkannt. So hat die Zeile

```
10 PRINT "LET"
```

die Ausgabe von LET zur Folge, ohne daß die Hintergrundfarbe verändert wird.

8.4. Mehrere zusätzliche Befehle

Will man mehrere Befehle einfügen, so wäre das Prinzip aus dem oberen Listing umständlich. Wollte man wie da jeden Befehl extra abfragen, so würde die Befehlserweiterung untragbare Ausmaße annehmen.
Es geht jedoch auch einfacher:

```
0380   INC $3D
0382   BNE $0386
0384   INC $3E
0386   STA $FF01
0389   LDY #$00
038B   LDA ($30),Y
0380   JMP $0B00
0390   STA $FF03
0393   NOP
-----------------------
0B00   LDY #$03
0B02   DEY
0B03   BMI $0B0C
0B05   CMP $0B23,Y
0B08   BEQ $0B17
0B0A   BNE $0B02
0B0C   CMP #$3A
0B0E   BCS $0B13
0B10   JMP $0390
0B13   STA $FF03
0B16   RTS
0B17   TYA
0B18   ASL
0819   TAY
0B1A   LDA $0B27,Y
0B1D   PHA
0B1E   LDA $0B26,Y
0B21   PHA
0B22   RTS

0B23   $40, $88, $FF

0B26   $2B, $0B
0B28   $43, $0B
0B2A   $53, $0B

0B2C   LDA #$00
0B2E   STA $FF00
```

```
0B31   LDA $D020
0B34   EOR #$FF
0B36   STA $D020
0B39   LDA $D021
0B3C   EOR #$FF
0B3E   STA $D021
0B41   JMP $0380
0B44   LDA #$00
0B46   STA $FF00
0B49   LDA $D021
0B4C   EOR #$FF
0B4D   STA $D021
0B51   JMP $0380
0B54   LDA #$00
0B56   STA $FF00
0B59   LDA $D020
0B5C   EOR #$FF
0B5E   STA $D020
0B61   JMP $0380
```

In dem Teil ab $0B00 werden diesmal drei Befehle definiert:

a) Pi: Funktion wie oben
b) LET: Funktion wie oben
c) Klammeraffe: Pi + LET

Das Prinzip ist ganz einfach: Die Codes der neuen Befehle werden einfach in einer Tabelle untergebracht, die Adressen, wo die neuen Befehle ausgeführt werden sollen, in einer anderen Tabelle. Schauen Sie sich nun das Listing einmal etwas intensiver an:

$0B00: Hier wird die Anzahl der neuen Befehle in das X-Register geladen.
$0B02: Und um eins verringert.
$0B03: Sind alle Befehle abgearbeitet, so wird zu Adresse $0B0C gesprungen. Durch diesen BRANCH-Befehl wird auch noch der Fall untersucht, ob das X-Register den Wert Null enthält (es wird allerdings auch erst bei dem Fall X=2 angefangen).

$0B05: Hier wird das eingegebene Zeichen mit einem Byte aus der Tabelle von $0B23 - $0B25 verglichen. Diese Tabelle enthält die Werte für die neuen Befehle.
$0B08: Hier wird zu der Adresse $0B17 gesprungen, falls das eingegebene Zeichen gleich einem der neuen Befehle war.
$0B0A: War es nicht gleich, so wird zu $0B02 gesprungen und somit das eingegebene Zeichen mit dem nächsten Befehl verglichen.
$0B0C: Bei dieser Adresse geht es weiter, wenn das eingegebene Zeichen mit keinem der neuen Befehle übereinstimmt. Bis Adresse $0B16 entspricht das dem Teil der CHRGET-Routine, der weggefallen war.
$0B17: Hier geht es weiter, wenn das eingegebene Zeichen mit einem Befehl übereinstimmte. Die Nummer des Befehls (sie kann von 0 - 2 gehen) wird in das X-Register geschoben,
$0B18: mit zwei multipliziert
$0B19: und wieder in das X-Register zurückgeschrieben.
$0B1A: Hier wird das High-Byte der Adresse, bei der der neue Befehl beginnt, in den Akkumulator geholt
$0B1D: und auf dem Stack abgelegt.
$0B1E: Hier wird das zugehörige Low-Byte geholt
$0B21: und auch auf dem Stack abgelegt.
Wie Sie sich schnell vergewissern können, werden bei den entsprechenden X-Register-Werten folgende Low/High-Bytes auf dem Stack abgelegt:

X-Reg.	Low-Byte	High-Byte	Befehl
0	$2B	$0B	KLammeraffe
1	$43	$0B	LET
2	$53	$0B	Pi

Diese Adresse muß übrigens immer ein Byte unter die Adresse zeigen, bei der die Routine wirklich beginnt. So beginnt der Befehl "LET" nicht bei $0B43, sondern bei $0B44. Der Grund wird beim nächsten Befehl klar:

$0B22: Hier wird ein Rücksprung aus einem Unterprogramm ausgeführt. Aber aus welchem Unterprogramm? $0B00 wurde doch mit JMP, nicht mit JSR aufgerufen! Nun, bei RTS wird einfach

die Rücksprungadresse vom Stack geholt, zuerst das Low-, dann das High-Byte. Eine Adresse wurde nun aber durch die vorhergehenden Befehle auf dem Stack abgelegt und in den Programm-Counter geholt. Dieser wird daraufhin erhöht, um zu dem nächsten Befehl zu kommen. Darum mußte die Adresse ein Byte unter die richtige Adresse zeigen. Zusammengefassend kann man also sagen, daß dem Computer "vorgemacht" wird, daß er auf Grund eines JSR-Befehls (der genau vor dem Anfang des jeweiligen Befehls liegt) zu der Adresse $0B22 gekommen ist.

```
$0B23 - $0B25: Tabelle mit Codes der neuen Befehle
$0B26 - $0B2B: Tabelle mit Adressen der neuen Befehle (-1!)
```

Ab $0B2C beginnen die neuen Befehle, die wohl nicht noch einmal erklärt werden müssen.

Aus dem Vorhergehenden sollten Sie nicht schließen, daß nur Befehle von einem Byte Länge oder schon bestehende Befehle verwendet werden können. Sie können natürlich auch eigene (längere) Befehle verwenden. Die Längen dieser Befehle können natürlich auch unterschiedlich sein. Realisieren können Sie das, indem Sie mit drei Tabellen arbeiten: Die erste enthält die Anfangsadressen der zu vergleichenden Strings, die zweite diese Strings und die dritte schließlich die Adressen der Befehle.

Noch ein Tip zum Schluß: Stellen Sie Ihren Befehlen immer ein Erkennungszeichen voran, z. B. den Klammeraffen. Dadurch läßt sich schnell eine Unterscheidung zwischen eigenen und Befehlen des Interpreters treffen, und die Ausführung geht schneller vonstatten.

9. BANKING

9. Banking

****

9.1. Theoretische Grundlagen

Wir wollen hier natürlich kein Theorie-Buch schreiben, aber ein paar Grundlagen über Speicherverwaltung sollten Sie sich schon aneignen.
Noch vor einigen Jahren waren Heimcomputer mit 64K, sei es ROM oder RAM, undenkbar. So hatte der VC-20 in der Grundversion nur 5K RAM, von denen 3583 Bytes mit BASIC-Programmen belegt werden konnten. Außerdem verfügte er über 20KByte ROM. Erweiterbar war das ganze bis auf 32K RAM und 24K ROM. Insgesamt wurden also 32K + 24K = 56K Speicher belegt. Diese konnten mit der eingebauten CPU (6502) problemlos adressiert werden, da diese über 16 Adreßleitungen verfügt und somit 2^16 = 65536 = 64K Speicher ansprechen kann.
Innerhalb kurzer Zeit wurden die Speicherbausteine jedoch so billig und so klein, daß auch Heimcomputer mit diesen bis an den Kragen vollgepropft werden konnten, ohne daß sie preislich aus der Spalte der Heimcomputer rutschten. Doch wie sollte der zusätzliche Speicher adressiert werden? Baute man eine CPU mit mehr Adreßleitungen ein, so würde der Preis sprunghaft ansteigen.
Da nicht nur der Heimcomputersektor unter diesem Problem zu leiden hatte, haben sich schlaue Leute hingesetzt und es entstanden drei Methoden, mit denen man auf mehr Speicher zugreifen kann, als adressierbar sind. Für uns ist nur eine dieser Methoden interessant, das sog. BANKING oder auch BANKSWITCHING. Bei dieser Methode werden einfach mehrere Speicherblöcke (im Englischen Banks, daher natürlich auch der Name), die oft den gesamten Raum des adressierbaren Speichers umfassen, "übereinander" angeordnet. Durch eine entsprechende Schaltung wird dann einfach zwischen diesen Banks umgeschaltet. Das ist für den Benutzer natürlich sehr einfach, stellt die Entwickler des Rechners jedoch vor einige Probleme. So kann es ohne weiteres vorkommen, daß beim Umschalten das Programm an einer Stelle weitergeführt

wird, die keinen sinnvollen Programmcode enthält. Der Rechner wird sich also "aufhängen". Besonders wichtig ist das beim Betriebssystem. Es bestünde natürlich die Möglichkeit, nicht immer den gesamten Speicherbereich umzuschalten, sondern einen Bereich, so z. B. das Betriebssystem, beizubehalten. Allerdings würde das natürlich heißen, das schon wieder Speicher verloren ginge, nämlich der Bereich, in dem der nicht wegblendbare Speicher liegt. Es gibt jedoch auch eine Möglichkeit, den gesamten Speicher umzuschalten: Man muß vorher sinnvolle Programmteile in den Speicher schreiben, der nach dem Umschalten benutzt werden soll. Das kann durch eine einfache Hardware-Lösung geschehen (das ist übrigens auch beim 64er der Fall; wenn Sie z. B. die Adresse $A000 lesen, so lesen Sie aus dem ROM, schreiben Sie jedoch darein, so schreiben Sie in das RAM).
Damit wären die theoretischen Grundlagen auch schon abgeschlossen und es wird zum 128er übergegangen.

9.2. Banking beim 128er

Wie man der Werbung von Commodore zum 128er entnehmen kann, besitzt dieser Computer 128KByte RAM (erweiterbar bis 512KByte) und 48KByte ROM. Da das nicht nebeneinander geht (jetzt zeigt es sich, ob Sie die theoretischen Grundlagen auch wirklich beherrschen), mußten wie beim C-64 Speicherblöcke übereinander gelegt werden. Dadurch ergibt sich folgendes Bild des Speichers:

(Zeichnung 9.2.a)

Doch wie wird dieser Speicher jetzt verwaltet?
Wie Sie aus dem Speicherplan schon ersehen konnten, arbeitet der C-128 wie der C-64 mit Banking (nur sind hier noch mehr Speicherblöcke übereinander). Um dieses Wirrwarr zu verwalten, benötigt man ein zusätzliches IC. Beim C-64 war das der sog. AM (adress manager). Da Commodore in der glücklichen Lage ist, eine eigene Halbleiterfirma zu besitzen (MOS technology), wurde flugs ein neues IC

entworfen, das sog. MMU (memory management unit). Dieses IC trägt die Nummer 8722. Es entscheidet, von welcher Bank der Benutzer den gewünschten Speicherinhalt erhält. Ja, der Benutzer, denn der Computer holt sich seine Informationen (z. B. für den BASIC-Interpreter) immer aus der richtigen Bank.

9.3. Umschalten der Banks über die MMU

Zum Umschalten der Banks benötigen wir nur ein Register der MMU, das Register 0. Es trägt den Namen CR (configuration register). Die einzelnen Bits haben folgende Funktion:

```
0     : Wählt den Bereich $D000 - $E000 aus:
        0 = I/O            1 = ROM / RAM
1     : Wählt den Bereich $4000 - $7FFF aus:
        0 = ROM            1 = RAM
2 & 3: Wählen den Bereich $8000 - $BFFF aus:
        00 = System ROM    01 = internal function ROM
        10 = external function ROM    11 = RAM
4 & 5: Wählen den Bereich $C000 - $FFFF aus:
        00 = System ROM    01 = internal function ROM
        10 = external function ROM    11 = RAM
6 & 7: Wählen den RAM-Speicher aus:
        00 = RAM-Bank 0    01 = RAM-Bank 1
        10 = RAM-Bank 2    11 = RAM-Bank 3
```

Dazu noch einige Erklärungen:

a) Bit 4 & 5 sind abhängig von Bit 0, d. h. wenn Bit 4 & 5 gesetzt sind, Bit 0 dagegen nicht, so besteht der Bereich von $C000 - $FFFF nicht vollständig aus RAM, wie man der Stellung von Bit 4 & 5 nach ja vermuten könnte, sondern von $D000 - $E000 befindet sich die I/O. RAM wäre da erst, wenn Bit 0 auf eins gesetzt wäre. Die Funktion von Bit 0 ist also folgende: ist es Null, so ist die I/O unabhängig von der Stellung der Bits 4 & 5 eingeschaltet. Ist es dagegen 1, so wird der Speicher von $C000 - $FFFF durch Bit 4 & 5 bestimmt.

b) Bit 6 & 7 ergeben bei den jetzigen Versionen des 128ers nur in den ersten beiden Stellungen einen Sinn, da die RAM-Bänke 2 & 3 noch nicht existieren.
c) Unabhängig von jeder Stellung dieses Registers befindet sich im Bereich von $FF00 - $FF04 ein Systemspeicher. Das erste dieser fünf Bytes ($FF00) stellt das Register 0 der MMU dar. Jetzt werden Sie sich fragen, warum das ganze? Ganz einfach: Stellen Sie sich vor, Sie hätten die I/O ausgeschaltet, um z. B. auf den Charactergenerator zugreifen zu können. Nun wollen Sie wieder auf sie zurückgreifen, um z. B. die Bildschirmfarbe zu ändern. Doch wie wollen Sie sie wieder einschalten? Die Register der MMU liegen schließlich auch in der I/O (Entschuldigung, wenn wir das bis jetzt vorenthalten haben): von $D500 - $D50B! Nun tritt Speicherstelle $FF00 in Aktion. Da dieses Register die gleiche Funktion wie das Register 0 hat, kann sofort wieder zurückgeschaltet werden.
Da Banking über Ändern von Registern nicht gerade benutzerfreundlich sein würde, haben die Programmierer des BASIC-Interpreters einen Befehl dazu implementiert, den BANK-Befehl. Die Syntax ist folgende:

BANK nr

"nr" stellt eine Zahl von 0 - 15 dar und setzt die dazu gehörige Funktion ingang.

```
Nr: Inhalt von $FF00 :       Eingeschalteter Speicher
-------------------------------------------------------
 0: $3F = %00111111  : RAM-Bank 0 durchgehend
 1: $7F = %01111111  : RAM-Bank 1 durchgehend
 2: $BF = %10111111  : RAM-Bank 2 durchgehend
 3: $FF = %11111111  : RAM-Bank 3 durchgehend
 4: $16 = %00010110  : Function-ROM intern, gemischt mit
                       RAM-Bank 0 und I/O
 5: $56 = %01010110  : wie oben, jedoch RAM-Bank 1
 6: $96 = %10010110  : wie oben, jedoch RAM-Bank 2
 7: $D6 = %11010110  : wie oben, jedoch RAM-Bank 3
 8: $2A = %00101010  : Function-ROM extern, gemischt mit
                       RAM-Bank 0 und I/O
```

```
 9: $6A = %01101010  : wie oben, jedoch RAM-Bank 1
10: $AA = %10101010  : wie oben, jedoch RAM-Bank 2
11: $EA = %11101010  : wie oben, jedoch RAM-Bank 3
12: $06 = %00000110  : Function-ROM intern low, mit Kernal,
                       RAM-Bank 0 und I/O
13: $0A = %00001010  : Function-ROM extern low, mit Kernal,
                       RAM-Bank 0 und I/O
14: $01 = %00000001  : Kernal mit BASIC, Zeichengenerator
                       und RAM-Bank 0
15: $00 = %00000000  : Kernal mit BASIC, I/O und RAM-Bank 0
```

Der Befehl "BANK 15" stellt die Konfiguration her, die nach dem Anschalten vorhanden ist.

Übertragen wir nun das ganze in die Praxis:
Geben Sie "BANK 15" ein, um das Kernal einzuschalten und rufen Sie nun mit SYS eine Routine darin auf, z. B. $FF4D (65357). Nach kurzer Zeit befinden Sie sich im 64er-Modus (die Routine wird im Kapitel KERNAL erklärt). Führen Sie nun einen Reset durch, geben Sie "BANK 0" ein und rufen Sie die Routine noch einmal auf. Der Rechner wird wahrscheinlich in den Monitor springen. Warum das? Durch BANK 0 haben Sie das Kernal abgeschaltet und der SYS-Befehl zielte auf das RAM (RAM-Bank 0), wo höchstwahrscheinlich kein sinnvoller Maschinen-Code vorlag. Sie müssen also in eigenen Programmen den BANK-Befehl verwenden, wenn Sie auf Speicherplätze zugreifen wollen, die normalerweise nicht eingeschaltet sind. In BASIC wird das jedoch wohl nicht so oft vorkommen.
Übrigens benutzt auch der BASIC-Interpreter beide RAM-Banks. In RAM-Bank 0 wird das Programm abgespeichert, in RAM-Bank 1 die zugehörigen Variablen. Das hat einige Vorteile. So werden z. B. die Variablen nicht verändert, wenn ein Programm unterbrochen und verändert wird (schauen Sie sich dazu auch einmal den Data-Generator in Kapitel ? an). Ein Nachteil jedoch ist, daß Programme nicht die vollen 128K belegen können.
Das war im großen und ganzen das Banking. Beispielprogramme, die sich das Umschalten zunutze machen, finden Sie über dieses Buch verstreut.

9.4. Weitere Möglichkeiten der MMU

Doch damit sind die Möglichkeiten der MMU bei weitem noch nicht ausgeschöpft. So kann man z. B. eine sog. "common area" programmieren. Dieser Speicherbereich ist (fast) frei wählbar in Größe und Lage und wird beim Umschalten von Banks beibehalten. So besteht eine einfach Möglichkeit, Daten von einer Routine in einer Bank zu einer anderen Routine in einer anderen Bank zu übergeben. Beim 128er sind nach dem Umschalten die Pages 0 - 3 common area, so daß bei Zugriffen darauf nicht umgeschaltet werden muß.
Für die common area sind die Bits 0 - 3 des Registers 6 der MMU zuständig. Dieses Register trägt den Namen RCR (RAM configuration register). Die Funktion der Bits 0 - 3 ist folgende:

```
0 & 1 : Größe der common area:
        00 =  1K   01 =  4K
        10 =  8K   11 = 16K
2 & 3 : Lage der common area:
        00 = keine   01 = unten
        10 = oben    11 = beides
```

Doch auch damit sind die Möglichkeiten noch nicht ausgeschöpft. Zusätzlich kann noch die Anfangsadresse der Page 0 (Zeropage) und die Anfangsadresse der Page 1 bestimmt werden. Das geschieht über die Register 7 - 10 der MMU:

```
  Register   :                Funktion
-------------------------------------------------------
      7      : Adreßbits  8 - 15 der Page 0
      8      : Adreßbits 16 - 19 der Page 0 (Bits 0 - 3)
      9      : Adreßbits  8 - 15 der Page 1
     10      : Adreßbits 16 - 19 der Page 1 (Bits 0 - 3)
```

Die Pages können logischerweise nur in 256Byte-Schritten verschoben werden.

Wie Sie sehen, bieten sich zahlreich Möglichkeiten für engagierte Programmierer. Einige Beispiele für Anwendungen dieser Features der MMU werden in diesem Buch gezeigt.

10. AUTOSTART

10. Autostart

****

10.1. Autostart mit der Floppy

Viele Leute, die heute mit Computern zu tun haben, besitzen keinerlei Programmiererfahrung. Für sie ist es oft schon schwer, ein Programm selbst zu starten. Bei größeren Rechnern ist das daher einfach. So existiert bei dem Betriebssystem MS-DOS die Möglichkeit, ein Programm sofort zu starten. Bei einem anderen bekannten Betriebssystem, CP/M, kann man ein Programm einfach durch Eingabe seines Namens aufrufen (das geht natürlich auch bei MS-DOS). Beim 64er war das schon schwieriger. Da mußte man LOAD, gefolgt von dem Namen (in Hochkommas / Gänsefüßchen) und einem ",8" (eventuell sogar ",8,1") eingeben. Nun muß man allerdings auch sagen, daß der C-64 nicht so sehr für reine Anwender, eher für Hobby-Programmierer , also für Leute mit Erfahrung im Umgang mit Computern, gedacht war. Der C-128 zielt da doch mehr auf Anwender, besonders da für ihn das Betriebssystem CP/M vorgesehen ist. Daher besitzt er auch eine Möglichkeit, Programme von Disk automatisch zu laden und auszuführen. Diese Routine wird auch bei Reset angesprungen, so daß der Rechner bei eingeschalteter Floppy und eingelegter Diskette nur angeschaltet werden muß, und das Programm wird in den Speicher geladen. Dabei gibt es zahlreiche Möglichkeiten. So kann man erst einmal beliebig viele Blöcke von Disk lesen (die allerdings fortlaufend angefangen bei Track 1, Sektor 1 auf der Diskette existieren müssen), bevor man ein Programm beliebigen Namens lädt. Nach dem Laden kann ein Programm ausgeführt werden. So könnte man also zuerst einmal eine neue Sprache (z. B. Forth) in den Speicher laden, dann ein Programm in der neuen Sprache nachladen (oder natürlich auch noch in BASIC) und dann die neue Sprache einschalten (oder erst das BASIC-Programm starten). Welche Möglichkeiten sich da auftun, wird sich wohl erst zeigen, wenn sich der C-128 eine Weile auf dem Markt befindet.

Im nächsten Kapitel wird die Routine, die für den Autostart sorgt, erklärt. Wenn Sie den Autostart nur anwenden wollen, so können Sie dieses Kapitel ruhig überschlagen. Wollen Sie dagegen tiefer in die Materie eindringen, so sollten Sie hier weiterlesen. Nur so können Sie diese Routine für eigene Zwecke benutzen.

10.1.1. Die Routine boot-call

Die Routine boot-call liegt im Betriebssystem von $F890 - $F98A. Aufgerufen werden kann sie auch über die Adresse $FF53, da sie auch in der Kernal-Sprung-Tabelle zu finden ist.

Zuerst einmal das Listing der Routine:

```
F890   STA $BF
F892   STX $BA
F894   TXA
F895   JSR $F23D
----------------------
F898   LDX #$00
F89A   STX $9F
F89C   STX $C2
F89E   INX
F89F   STX $C1
----------------------
F8A1   INY
F8A2   BNE $F8A1
F8A4   INX
F8A5   BNE $F8A1
---------------------
F8A7   LDX #$0C
F8A9   LDA $FA08,X
F8AC   STA $0100,X
F8AF   DEX
F8B0   BPL $F8A9
```

```
----------------------
    F8B2   LDA $BF
    F8B4   STA $0106
    F8B7   LDA #$00
    F8B9   LDX #$0F
    F8BB   JSR $F73F
    F8BE   LDA #$01
    F8C0   LDX #$15
    F8C2   LDY #$FA
    FAC4   JSR $F731
    F8C7   LDA #$00
    F8C9   LDY #$0F
    F8CB   LDX $BA
    F8CD   JSR $F738
    F8D0   JSR $FFC0
    F8D3   BCS $F8EB
----------------------
    F8D5   LDA #$01
    F8D7   LDX #$16
    F8D9   LDY #$FA
    F8DB   JSR $F731
    F8DE   LDA #$0D
    F8E0   TAY
    F8E1   LDX $BA
    F8E3   JSR $F738
    F8E6   JSR $FFC0
    F8E9   BCC $F8EE
----------------------
    F8EB   JMP $F98B
----------------------
    F8EE   LDA #$00
    F8F0   LDY #$0B
    F8F2   STA $AC
    F8F4   STY $AD
----------------------
    F8F6   JSR $F9D5
----------------------
    F8F9   LDX #$00
    F8FB   LDA $0B00,X
    F8FE   CMP $E2C4,X
```

```
F901   BNE $F8EB
F903   INX
F904   CPX #$03
F906   BCC $F8FB
----------------------
F908   JSR $FA17
F90E   4F 54 49 4E
F912   47 20 00
----------------------
F915   LDA $0B00,X
F918   STA $A9,X
F91A   INX
F91B   CPX #$07
F91D   BCC $F917
----------------------
F91F   LDA $0B00,X
F922   BEQ $F92A
F924   JSR $FFD2
F927   INX
F928   BNE $F91F
F92A   STX $9E
----------------------
F92C   JSR $FA17
F92F   2E 2E 2E 0D
F933   00
----------------------
F934   LDA $AE
F936   STA $C6
----------------------
F938   LDA $AF
F93A   BEQ $F945
F93C   DEC $AF
F93E   JSR $F9B3
F941   INC $AD
F943   BNE $F938
F945   JSR $F98B
----------------------
F948   LDX $9E
F94A   .BYTE $2C
F94B   INC $9F
```

```
F94D   INX
F94E   LDA 0B00,X
F951   BNE $F94B
F953   INX
F954   STX $04
-----------------------
F956   LDX $9E
F958   LDA #$3A
F95A   STA $0B00,X
F95D   DEX
F95E   LDA $BF
F960   STA $0B00,X
F963   STX $9E
-----------------------
F965   LDX $9F
F967   BEQ $F97E
F969   INX
F96A   INX
F96B   TXA
F96C   LDX $9E
F96E   LDY #$0B
F970   JSR $F731
F973   LDA #$00
F975   TAX
F976   JSR $F73F
F979   LDA #$00
F97B   JSR $F269
-----------------------
F97E   LDA #$0B
F980   STA $03
F982   LDA #$0F
F984   STA $02
F986   JSR $02CD
-----------------------
F989   CLC
F98A   RTS
```

Mit den ersten beiden Befehlen wird der Inhalt des Akkus und des X-Registers in Adressen in der Zeropage gespeichert. Um es nicht zu spannend zu machen, wird hier gleich die Funktion der beiden Bytes genannt: Der Akkumulator stellt die Geräteadresse der Floppystation dar, von der nachher eventuell ein Programm geladen wird. Das X-Register enthält die Geräteadresse des Laufwerkes, auf dem die eingelegte Diskette nach einem Autostart abgesucht werden soll. In den meisten Fällen werden diese beiden Bytes identisch sein. Bei einem Reset sind beide Registerinhalte auf 8 gesetzt, ein Autostart nach dem Einschalten wird also nur auf der Floppy mit der Geräteadresse 8 durchgeführt. Sie können die Routine natürlich auch anders anspringen, z. B. mit "SYS DEC("FF53"),9,9". Hier wird auf einer angeschlossenen Floppy mit der Geräteadresse 9 nach der Möglichkeit eines Autostartes gesucht.
Das Unterprogramm, das in Adresse $F895 aufgerufen wird, schließt alle Files mit der Geräteadresse, die sich bei Ansprung der Routine im Akku befinden. Die Funktion ist klar: Die Befehle und Daten, die nachher zu der Floppy gesendet werden sollen, würden von der Floppy nicht bzw. zu spät übernommen werden, wenn eine andere offene Datei noch die Floppy belegt. Das Schließen geschieht übrigens auf ziemlich einfache Weise: Zuerst wird in einer Tabelle, die die Geräteadressen der gerade offenen Dateien enthält ($036C - $0375), nach der angegebenen Gerätenummer gesucht. Stößt der Rechner in dieser Tabelle auf die Gerätenummer, so holt er aus einer anderen Tabelle ($0362 - $036B) die dazugehörige Dateinummer (die Einträge in den beiden Tabellen stehen immer an der gleichen Platznummer). Diese Datei wird dann mit der Kernal-Routine CLOSE ($FFC3) geschlossen. Dieser Vorgang geschieht mit allen offenen Dateien (die Anzahl steht in $98). Es existiert übrigens noch eine dritte dazugehörige Tabelle ($0376 - $037F). Diese enthält die Sekundäradressen der geöffneten Dateien (auch wieder in der gleichen Reihenfolge wie in den beiden anderen Tabellen).
Die folgenden fünf Befehle der boot-call-Routine setzen folgende Bytes der Zeropage auf folgenden Inhalt:

```
$9F (159): 00
$C1 (193): 01
$C2 (194): 00
```

Die Adresse $9F stellt einen Zähler dar und wird später noch benutzt.
In $C1 kommt die Track-, in $C2 die Sektornummer des Blocks auf der Diskette, aus dem später die Daten geholt werden. Die Daten für boot-call belegen also Track 1, Sektor 0, den ersten Block auf der Diskette. Dieser Block darf also vorher nicht anderweitig belegt worden sein. Am idealsten ist es, wenn Sie nur "frisch" formatierte Disketten mit einem Autostart versehen. Sie können sich natürlich auch mit einem Diskmonitor vergewissern, ob dieser Block belegt ist.
Das gerade Track 1, Sektor 0 ausgewählt wurde, ist übrigens nicht nur bei der Autostart-Routine in Gebrauch, sondern auch bei dem Betriebssystem CP/M. Auch dieses Programm benutzt den Autostart und meldet sich so sofort, wenn die CP/M-Diskette bei Anschalten des Rechners in der Floppy liegt. Doch weiter bei boot-call. Die nächsten vier Befehle (von $F8A1 - $F8A6) stellen eine Warteschleife dar.
Dann werden 13 Bytes aus dem Betriebssystem (von $FA08 - $FA14) in das RAM (von $0100 - $010C) kopiert. Die Bytes stellen folgende Werte dar:

```
FA08: 30 30 20 31 30 20 30 20 33 31 3A 31 55
```

Und der Sinn des ganzen? Übersetzt man die Zeichen in ASCII-Format und ordnet sie von hinten nach vorne an (so werden sie später auch zur Floppy gesendet), so ergibt sich folgende Meldung:

```
U1:13 0 01 00"
```

Wenn Sie sich mit den Floppy-Befehlen auskennen, so wird Ihnen der Sinn dieser Befehlssequenz klar sein. Wenn Sie sich nicht mit den Befehlen auskennen, so müssen Sie sich noch ein bißchen Gedulden.
Die beiden Befehle von $F8B2 - $F8B6 speichern den Inhalt von $BF in den String. Die Adresse $BF wurde ganz zu Anfang

der Routine mit der gewünschten Geräteadresse geladen, d. h. anstelle der einzelnen Null in dem String wird die Geräteadresse geschrieben.
Die nächsten 11 Befehle haben zusammengefaßt folgende Funktion:

Wert	Kommt nach	Funktion
#00	$C6	Bank-Nr für LOAD/SAVE/VERIFY
#0F	$C7	Bank-Nr für aktuellen Filenamen
#01	$B7	Länge des aktuellen Filenamens
#15	$BB	Low-Byte Adresse des Filenamens
#FA	$BC	High-Byte Adresse des Filenamens
#00	$B8	Logische Dateinummer
#0F	$B9	Sekundäradresse
$BA	$BA	Geräteadresse

Der Filenamen steht demnach an der Adresse $FA15. In dieser Adresse steht der Wert $49 (73). Dieser Wert ist als ASCII-Wert zu interpretieren und stellt das Zeichen "I" dar. Der Befehl danach (JSR $FFC0) ruft die Kernal-Routine OPEN auf. Diese Assembler-Befehle stellen also folgenden BASIC-Befehl dar:

```
OPEN 0,Geräteadresse,15,"I"
```

Das "I" dahinter steht für den Diskettenbefehl "INITIALIZE". Dieser Befehl liest das BAM der gerade eingelegten Floppy. Dieser Befehl muß nie gebraucht werden, wenn Sie Ihren Disketten immer unterschiedliche IDs geben, da in diesem Fall die BAM automatisch von der Floppy eingelesen wird. Da sich die Programmierer des C-128er Betriebssystemes nicht darauf verlassen wollten (haben Ihre Disketten alle eine unterschiedlich ID?), haben sie einfach diesen Befehl eingefügt.
Über die Datei mit der Nummer Null können nun also Befehle zur Floppy gesendet werden.
In Adresse $F8D3 steht ein BRANCH-Befehl. Durch ihn wird zur Adresse $F8EB gesprungen, wenn sich die Floppy nicht gemeldet hat (weil sie z. B. nicht angeschaltet war). Es

wird in diesem Fall keine Fehlermeldung ausgegeben. Probieren Sie es aus (Sie werden Ihre Floppy wohl kaum auf die Geräteadresse 10 umgestellt haben):

```
SYS DEC("FF53"),10,10
```

Es erscheint einfach wieder "READY.".

Wenn sich die Floppy gemeldet hat, geht es weiter in Adresse $F8D5. Dort werden wieder einige Adressen gesetzt:

Wert	Kommt nach	Funktion
#01	$B7	Länge des aktuellen Filenamens
#16	$BB	Low-Byte Adresse des Filenamens
#FA	$BC	High-Byte Adresse des Filenamens
#00	$B8	Logische Dateinummer
#00	$B9	Sekundäradresse
$BA	$BA	Geräteadresse

In der Adresse $FA16, auf die der Zeiger $BB/$BC gerichtet ist, steht der Wert $23 (35). Der Filename ist demnach "#".
Zusammenfassend entsprechen die Assembler-Befehle (mit dem Aufruf der OPEN-Routine in $F8E6) folgendem BASIC-Befehl:

```
OPEN 13,Geräteadresse,13,"#"
```

Falls Sie sich nicht mit der Floppy auskennen: Dadurch wird in der adressierten Floppy ein Puffer für die Daten eingerichtet, die später von der Floppy gelesen werden. Auf diesen Puffer kann über den Kanal 13 zugegriffen werden.
Auch hier wird die boot-call-Routine wieder verlassen, wenn sich die Floppy nicht meldet.

Die vier Befehle von $F8EE - $F8F5 setzen die Bytes $AC/$AD auf $0B00 (Kassettenpuffer). Diese beiden Zeropage-Adressen werden gleich als Zeiger benutzt; in den Kassettenpuffer werden die von der Floppy gelesenen Zeichen gespeichert.
In Adresse $F8F6 wird eine Meldung zur Floppy ausgegeben und daraufhin Zeichen geholt. Diese Routine ist ziemlich wichtig

und wird deswegen hier noch extra aufgelistet (diesmal aber gleich dokumentiert):

```
F9D5   LDX #$00      ; Logische Filenummer
F9D7   JSR $FFC9     ; Ausgabegerät mit Geräteadresse
                       der Datei mit der Nummer 0 setzen
F9DA   LDX #$0C      ; Anzahl der auszugebenden Zeichen - 1
F9DC   LDA $0100,X   ; Zeichen holen
F9DF   JSR $FFD2     ; und ausgeben (BSOUT)
F9E2   DEX           ; Alle Zeichen?
F9E3   BPL $F9DC     ; nein
F9E5   JSR $FFCC     ; Schließen des gerade aktiven I/O-
                       Kanals am IEC-Bus (CLRCH)
F9E8   LDX #$0D      ; Logische Filenummer
F9EA   JSR $FFC6     ; Eingabegerät mit Geräteadresse
                       der Datei mit der Nummer 13 setzen
F9ED   LDY #$00      ; Zähler auf Null
F9EF   JSR $FFCF     ; Zeichen eingeben (BASIN)
F9F2   JSR $F7BC     ; und in Adresse speichern, die durch
                       $AC/$AD angegeben wird (in Bank, die
                       durch $C6 bestimmt wird)
F9F5   INY           ; Offset für Zeiger $AC/$AD erhöhen
F9F6   BNE $F9EF     ; noch nicht alle Zeichen (1 Block)
F9F8   JMP $FFCC     ; Gerade aktiven I/O-Kanal am IEC-Bus
                       schließen und zurückspringen
```

In diesem Fall wird die Meldung "U1:13 0 01 00" ausgegeben und dann der Block Track 1, Sektor 0 in den Puffer geholt. Dieser Buffer wird dann ausgelesen und die Bytes von $0B00 - $0BFF werden abgespeichert.
Dann geht es in der boot-call-Routine weiter.
Hier werden die ersten drei eingelesenen Bytes mit den Inhalten der Adressen $E2C4 - $E2C6 verglichen. Ist eines dieser Bytes nicht identisch, so wird die Routine an der gleichen Stelle verlassen, als ob sich die Floppy nicht gemeldet hätte. Das heißt, diese drei Bytes enthalten die Information, die darüber entscheidet, ob ein Autostart ausgeführt wird oder nicht!
Die drei Bytes im Betriebssystem haben folgende Werte:

```
$E2C4 : $43 (67)
$E2C5 : $42 (66)
$E2C6 : $4D (77)
```

Die ersten drei Bytes von Track 1, Sektor 0 stellen zusammen den String "CBM" dar.
Stimmen alle Bytes überein, so wird "(CR)BOOTING " ausgegeben. Das geschieht mittels der Kernal-Routine primm (siehe Kapitel Kernal).
Anschließend werden die Inhalte der Adressen $0B03 - $0B06 in die Adressen $AC - $AF kopiert. Die Inhalte der Adressen $0B03 - $0B06 entsprechen den Bytes 3 - 6 des von der Diskette geholten Blockes. Zur der Funktion dieser Bytes kommen wir später noch.
Im nächsten Abschnitt werden ab $0B07 (also dem siebten Byte des Blockes) bis zur nächsten Null die Werte als ASCII-Werte interpretiert und die dazugehörigen Zeichen ausgegeben. Die Stelle, an der die Null auftrat, wird anschließend in der Adresse $9E gespeichert.
Dann wird wieder die primm-Routine aufgerufen und der Text "...(CR)" ausgegeben.
In den nächsten beiden Befehlen wird der Inhalt der Adresse $AE nach $C6 kopiert. Die Adresse $AE wurde weiter oben mit dem Inhalt von $0B05 geladen, demnach also mit dem Byte 5 des gelesenen Blockes. Die Adresse $C6 wurde auch schon einmal benutzt, nämlich in dem weiter oben dokumentierten Unterprogramm. Dort wurde mit dem Inhalt von $C6 die Bank bestimmt, in die der zu ladende Block gespeichert werden sollte.
Im nächsten Abschnitt wird der Inhalt von $AF geholt. Ist er von Null verschieden, so wird der Wert um eins vermindert. Ist er gleich Null, so wird direkt zu einem weiteren Unterprogramm gesprungen:

```
F9B3   LDX $C2      ; Sektor-Nr. (bei ersten Ansprung 0)
F9B5   INX          ; Um eins erhöhen
F9B6   CPX #$15     ; Schon Sektor 21?
F9B8   BCC $F9BE    ; nein
F9BA   LDX #$00     ; ja, dann Sektor = 0
F9BC   INC $C1      ; und Track-Nr. um eins erhöhen
```

```
F9BE   STX $C2      ; Sektor-Nr. wieder abspeichern
F9C0   TXA          ; Sektor-Nr. in Akku
F9C1   JSR          ; Sektor-Nr in ASCII-Zeichen umwandeln
F9C4   STA $0100    ; Unteres Nibble (1er-Ziffer)
F9C7   STX $0101    ; Oberes Nibble (10er-Ziffer)
F9CA   LDA $C1      ; Track-Nr. holen
F9CC   JSR $F9FB    ; und in ASCII-Zeichen umwandeln
F9CF   STA $0103    ; Unteres Nibble (1er-Ziffer)
F9D2   STA $0104    ; Oberes Nibble (10er-Ziffer)
-------------------------------------------------------
F9D5 - F9FA wurde schon oben dokumentiert
-------------------------------------------------------
F9FB   LDX #$30     ; Oberes Nibble = "0"
F9FD   SEC          ; Carry-Flag für Subtraktion setzen
F9FE   SBC #$0A     ; Zahl durch Ziffern 0-9 darstellbar?
FA00   BCC $FA05    ; ja, also Ende
FA02   INX          ; nein, also 10er-Ziffer erhöhen
FA03   BCS $F9FE    ; und noch einmal versuchen
FA05   ADC #$3A     ; Unteres Nibble in ASCII umwandeln
FA07   RTS
```

Das Unterprogramm, das Zahlen in ASCII-Werte umwandelt, funktioniert übrigens nur bei Zahlen, die dezimal mit zwei Ziffern darstellbar sind. Denken Sie daran, wenn Sie die Routine für eigene Zwecke benutzen.

Was ist - kurz gesagt - die Funktion der boot-call-Routine? Der Inhalt von $AF bestimmt, wieviele Blöcke noch von Diskette gelesen werden sollen, $AC/$AD bestimmt, ab welcher Adresse die Blöcke gespeichert werden sollen (sie können also nicht Pages überspringen, die Blöcke werden immer fortlaufend im Speicher untergebracht) und $AE gibt die Bank an, in die die Blöcke gespeichert werden sollen. Die Blöcke werden fortlaufend ab Track 1, Sektor 1 gelesen (als nächstes dann Track 1, Sektor 2 usw.).
Da die Inhalte der Adressen $AC - $AF durch die Bytes 3 - 7 des ersten gelesenen Blockes bestimmt werden, können Sie durch diesen Block weitere Aufgaben angeben.
Aber es geht noch weiter:
Unabhängig davon, ob noch weitere Blöcke gelesen wurden,

oder nicht, werden in Adresse $F945 die gerade geöffneten Dateien geschlossen.
Dann wird der Zähler wiedergeholt. Welcher Zähler? Wie Sie sich vielleicht erinnern, war in $9E die Stelle abgespeichert worden, bei der der ausgegebene String zu Ende war. Dieser Zähler wird erhöht und das Zeichen, auf das er nun zeigt, geholt. Ist es Null, so wird dieser Abschnitt der boot-call-Routine verlassen. Ist es nicht Null, so wird ein Zeiger (der ganz zu Anfang auf Null gesetzt worden war) um eins erhöht. Bei Null wird der Zeiger noch einmal erhöht (er zeigt dann also auf das erste Zeichen nach der Null) und dann in $04 gespeichert.
Im nächsten Abschnitt wird der Zeiger $9E, der immer noch auf die erste Null nach dem auszugebenden Text zeigt, als Offset benutzt. An die Stelle der ersten Null wird der ASCII-Wert für den Doppelpunkt geschrieben und davor kommt der Inhalt von $BF (wurde durch Akku bei Einsprung von boot-call gesetzt). Der neue Zähler wird nun gespeichert.
Danach wird die Länge des Filenamens in das X-Register geholt. Ist sie Null, so wird dieser Teil gleich übersprungen, da demnach kein weiteres Programm eingelesen werden soll. Ist sie jedoch ungleich Null, so wird sie um zwei erhöht, da ja noch der Doppelpunkt und der Inhalt von $BF dazugekommen sind. Anschließend werden wieder einige Adressen gesetzt:

Wert	Kommt nach	Funktion
Akku	$B7	Länge des aktuellen Filenamens
$9E	$BB	Low-Byte Adresse des Filenamens
#0B	$BC	High-Byte Adresse des Filenamens
#00	$Cc	Bank-Nr. für LOAD/SAVE/VERIFY
#00	$C7	Bank-Nr. für aktuellen Filenamen

In Adresse $F979 wird der Akkumulator mit Null geladen, um zu kennzeichnen, daß LOAD gemeint ist. Denn das im nächsten Befehl aufgerufene Unterprogramm wird auch bei VERIFY benutzt. Dann wäre das Flag jedoch 1.
Anschließend wird das Programm geladen.
Bei Adresse $F97E geht es dann weiter, egal ob ein Programm geladen wurde, oder nicht.

Hier wird zuerst in die Adresse $0004 der Wert $0B, dann in $0002 der Wert $0F gespeichert. Daraufhin wird ein Unterprogramm angesprungen, das auch von einer Kernal-Routine benutzt wird (bzw. zum größten Teil diese Kernal-Routine darstellt). Dieser Aufruf macht folgendes: es wird die Bank eingestellt, deren Nummer in $0002 steht (hier also Bank 15 und damit die Normal-Konfiguration). Dann wird zu der Adresse gesprungen, die durch $0003/$0004 gegeben ist (in unserem Fall ein Zeiger auf das erste Byte hinter der Null hinter dem Programmnamen). Dieses Programm wird ausgeführt. Soll dieses Unterprogramm durch ein RTS wieder verlassen werden, so wird hinter dem Programmaufruf in der Bank, von der aus aufgerufen wurde, weitergemacht (hier also bei Adresse $F989).
In Adresse $F989 wird das Carryflag gelöscht. Damit wird angezeigt, daß die boot-call-Routine ordnungsgemäß verlassen wurde (hatte sich die Floppy nicht gemeldet, so wurde das Carryflag gesetzt, siehe Adresse $F8D3).
Schließlich wird die Routine durch ein RTS verlassen.

10.1.2. Verwendung von boot-call

Bei der Untersuchung der boot-call-Routine im vorigen Kapitel wurde der Schluß gezogen, daß der erste Block auf einer Diskette, Track 1 / Sektor 0, folgende Byte-Belegung haben muß:

```
1. Byte: $43 (67) ASCII-Wert für "C"
2. Byte: $42 (66) ASCII-Wert für "B"
3. Byte: $4D (77) ASCII-Wert für "M"
4. Byte: Low-Byte der Adresse, ab der weitere Blöcke
         gespeichert werden (kommt in Adresse $AC)
5. Byte: High-Byte dieser Adresse (kommt nach $AD)
6. Byte: Bank-Nummer, in die die weiteren Blöcke
         gelesen werden (kommt nach $AE)
7. Byte: Anzahl der weiter zu lesenden Blöcke (kommt
         nach $AF)
-------------------------------------------------------
8. Byte: Bis zur ersten Null: Text, der hinter BOOTING
         ausgegeben wird
         Hinter  der ersten Null bis zur zweiten:  Name
des
         Programms, das nach dem Einlesen der Blöcke
         noch geladen werden soll
         Hinter  der  zweiten Null  bis  höchstens  zum
Ende:
         Programm, das nach dem Laden noch ausgeführt
         werden soll
```

Probieren wir die neu gewonnenen Erkenntnisse einmal aus:

```
10 OPEN 1,8,15
20 OPEN 2,8,13,"#"
25 PRINT#1,"B-F 0 1 0"
30 PRINT#1,"B-P 13 0"
40 PRINT#2,"CBM"
50 PRINT#1,"U2 13 0 1 0"
60 PRINT#1,"B-A 0 1 0"
70 CLOSE 2
80 CLOSE 1
```

Wer sich mit den Floppy-Befehlen nicht auskennt: Ersteinmal wird eine Datei geöffnet, mit der man auf den Befehlskanal der Floppy zugreifen kann, danach eine Datei, mit der man auf einen Puffer innerhalb der Floppy zugreifen kann. Daten können nämlich nicht direkt auf die Disketten geschrieben werden, sondern nur über einen Datenpuffer.
In Zeile 20 wird der Block, der gleich beschrieben werden soll, als frei gekennzeichnet. Vorsicht! Ein Programm, das eventuell auf diesem Block lag, wird dadurch gelöscht!
In Zeile 30 wird ein Zeiger auf das erste Byte (Byte 0, nicht 1!) dieses Puffers gesetzt. Danach wird in diesen Puffer die Zeichenfolge "CBM" geschrieben. Mit dem Floppy-Befehl in Zeile 50 wird dieser Datenpuffer (mit der Kanalnummer 13) auf die Diskette geschrieben und zwar auf den Block Track 1 / Sektor 0. Schließlich muß dieser Block noch als belegt gekennzeichnet werden und dann werden die Dateien wieder geschlossen. Falls Sie die Befehle noch nicht genau verstanden haben, so schauen Sie einmal in Ihrem Floppy-Handbuch nach.

Rufen Sie nun die Routine boot-call auf:

```
SYS DEC("FF53"),8,8  (oder auch SYS DEC("F890"),8,8)
```

Die Floppy-Station läuft an und nach kurzer Zeit wird der Text "BOOTING..." ausgegeben. Danach wird der Computer wahrscheinlich in den Monitor springen. Warum das? Nun, wir haben nur die ersten drei Bytes des Blocks belegt, nur die Bytes, die den Autostart "genehmigen". Der Rest bestand wahrscheinlich aus Nullen, so daß der Computer keinen weiteren Text ausgab, und auch nach keinen weiteren Blöcken oder Programmen suchte. Danach wollte er ein Maschinenprogramm ausführen, fand jedoch nur den Code 0, was dem Maschinensprache-Befehl BRK entspricht und sprang so in den Monitor (die übliche Reaktion des C-128 auf den BREAK-Befehl).

Geben wir also ein paar mehr Werte an:

```
10 DIM BY(255)
20 BY(0)=ASC("C"): BY(1)=ASC("B"): BY(2)=ASC("M")
30 BY(3)=0: BY(4)=0: BY(5)=0: BY(6)=0
40 PRINT CHR$(147);"TEXT HINTER 'BOOTING':"
50 FOR ZA=7 TO 252
60 GET KEY A$: IF A$=CHR$(13) THEN 80
70 BY(ZA)=ASC(A$): NEXT ZA
80 BY(ZA)=0
90 ZA=ZA+1: BY(ZA)=0
100 ZA=ZA+1: BY(ZA)=96
110 OPEN 1,8,15
120 OPEN 2,8,13,"#"
125 PRINT#1,"B-F 0 1 0"
130 PRINT#1,"B-P 13 0"
140 FOR ZA=0 TO 255
150 PRINT#2,CHR$(BY(ZA));
160 NEXT ZA
170 PRINT#1,"U2 13 0 1 0"
180 PRINT#1,"B-A 0 1 0"
190 CLOSE 2
200 CLOSE 1
```

Hier wird zuerst ein Feld mit 256 Elementen angelegt. Die ersten 7 Bytes dieses Feldes (das natürlich den später zu schreibenden Block darstellt) werden so gesetzt, das ein Autostart erkannt werden kann und zwar ohne das weitere Blöcke geladen werden. Dann können Sie einen beliebigen Text eingeben, der jedoch nicht länger als 246 Zeichen sein darf, da noch andere Bytes gesetzt werden müssen. Das geschieht in den folgenden Zeilen:

```
Zeile  80: Endekennzeichen des auszugebenden Strings
Zeile  90: Endekennzeichen des zu ladenden Programms
           (da kein weiteres Zeichen dazwischen liegt,
            wird kein Programm geladen)
Zeile 100: Auszuführundes Programm (hier nur ein
            Befehl: RTS)
```

Dann werden die Bytes auf die Diskette geschrieben.

Starten Sie nach dem Ablauf des Programms wieder die boot-call-Routine. Jetzt wird ihr Text ausgegeben und dann wieder zurückgesprungen.

Nun eine Anwendung des Autostartes. Wenn Sie das folgende Programm starten, so wird der Block Track 1 / Sektor 0 so beschrieben, daß ein (BASIC-)Programm namens "HELLO" automatisch gestartet wird.
Die Belegung der einzelnen Bytes muß dann folgendermaßen aussehen:

```
Byte  0 -  6: Wie gehabt
Byte  7 - 26: Auszugebender Text
Byte 27     : Null als Endekennzeichen
Byte 28 - 32: Name des zu ladenden Programms ("HELLO")
Byte 33     : Null als Endekennzeichen
Byte 34 - 52: Auszuführendes Maschinenprogramm (RUN)
```

Nun das Programm, das diese Belegung herstellt:

```
10 OPEN 1,8,15
20 OPEN 2,8,13,"#"
25 PRINT#1,"B-F 0 1 0"
30 PRINT#1,"B-P 13 0"
40 FOR ZA=0 TO 52
50 READ A
60 PRINT#2,CHR$(A);
70 NEXT ZA
80 PRINT#1,"U2 13 0 1 0"
90 PRINT#1,"B-A 0 1 0"
100 CLOSE 2
110 CLOSE 1
1000 DATA  67, 66, 77,  0,  0,  0,  0
1010 DATA  13, 40, 67, 41, 32, 49, 57, 56, 53, 32, 66
1020 DATA  89, 32, 87, 69, 82, 78, 69, 82, 32
1030 DATA   0
1040 DATA  72, 69, 76, 76, 79
1050 DATA   0
1060 DATA 162,  2,189, 50, 11,157, 74,  3,202, 16,247
1070 DATA 169,  3,133,208, 96, 82,213, 13
```

Das Assemblerprogramm (Datas in den Zeilen 1060 & 1070) sieht folgendermaßen aus:

```
        1  LDX #$02          ; Zähler
    loop 2  LDA tabelle,X    ; Zeichen holen
        3  STA $034A,X       ; und in Tastaturpuffer
        4  DEX               ; Noch ein Zeichen?
        5  BPL loop          ; ja
        6  LDA #$03          ; Anzahl der Zeichen
        7  STA $D0           ; im Tastaturpuffer
        8  RTS               ; Ende der Routine
 tabelle 9  $52, $D5, $0D    ; ASCII-Werte für R, Shift + U
                             ; und RETURN
```

Die Routine speichert also RUN in den Tastaturpuffer.

Nachdem Sie Ihre Diskette mit dem oben stehenden Programm behandelt haben, rufen Sie noch einmal die boot-call-Routine auf. Vorschriftsmäßig wird der Text ausgegeben, dann jedoch mit einem "FILE NOT FOUND" zurückgesprungen. Na klar, als Filename wurde ja "HELLO" angegeben, ohne daß sich dieses Programm auf der Diskette befand.
Wie könnte dieses Programm aussehen? Zum Beispiel so:

```
10 PRINT CHR$(147)
20 PRINT"TREFFEN SIE EINE AUSWAHL:"
30 PRINT: PRINT"(1) DIRECTORY"
40 PRINT: PRINT"(2) PROGRAMM LADEN"
50 PRINT: PRINT"(3) DISKETTEN-OPERATION"
60 PRINT: PRINT: PRINT"(4) BEENDEN"
70 GET KEY A$: A=VAL(A$)
80 ON A GOTO 100,200,300,400
90 GOTO 70
100 PRINT CHR$(147)
110 DIRECTORY
120 PRINT: PRINT"- BITTE TASTE DRUECKEN -"
130 GET KEY A$: GOTO 10
```

```
200 PRINT CHR$(147)
210 INPUT "NAME DES PROGRAMMS";NA$
220 LOAD NA$,8,1
300 PRINT CHR$(147)
310 INPUT "BEFEHL";BF$
320 OPEN 1,8,15
330 PRINT#1,BF$
340 CLOSE 1
350 GOTO 10
400 PRINT CHR$(147)
410 PRINT"SIND SIE SICHER (J/N)?"
420 GET KEY A$: IF A$="N" THEN 10
430 IF A$<>"J" THEN 420
```

Hier haben Sie verschiedene Möglichkeiten, die wohl nicht näher erklärt werden müssen.

Es sind natürlich auch ganz andere Möglichkeiten denkbar. So können Sie z. B. den Namen des Programms, das von boot-call geladen werden soll, so umändern, das automatisch Ihr Lieblingsspiel geladen wird (oder natürlich auch ein Nutzprogramm, das Sie benutzen).

Bei der boot-call-Routine existiert auch die Möglichkeit, daß noch mehrere Blöcke nachgeladen werden. Dadurch können Sie z. B. ein anderes Betriebssystem oder eine andere Sprache nachladen etc, etc.
Die entsprechenden Daten müssen sich jedoch fortlaufend auf der Diskette befinden (angefangen bei Track 1 / Sektor 1) und sie werden auch fortlaufend in den Speicher geladen. Die Bytes, die über dieses Nachladen entscheiden, sind die Bytes 3 - 6 auf dem Block Track 1 / Sektor 0 (siehe oben).

Diese Routine wird wohl oft eingesetzt werden, wahrscheinlich bei jedem guten Nutzprogramm (z. B. Textverarbeitung). Sie können sie jedoch genauso gut für Heimzwecke benutzen.

10.2. Autostart bei Modulen

Wer kennt sie nicht, die kleinen Schachteln, die hinten in den Expansionsport gesteckt werden und sich beim Einschalten des Rechners automatisch melden? Sie werden Module oder auch Cartridges genannt.
Der automatische Start hat hier zwei Vorteile:

1. Er ist benutzerfreundlich, da ein Anwender keine umständlichen SYS-Befehle eintippen muß.
2. Das Modul wird dadurch schwerer kopierbar.

Doch wie funktioniert dieser Autostart?

Beim 64er ging das folgendermaßen:
Bei einem Reset schaute der Rechner die Speicherstellen $8004 - $8008 nach. Fand er dort die ASCII-Werte der Zeichenfolge "CBM80", so führte er einen indirekten Sprung über die Adressen $8000 / $8001 durch, d. h. er holte sich den Inhalt von $8000 als Low- und den Inhalt von $8001 als High-Byte. Zu dieser Adresse wurde dann gesprungen.

Beim 128er geht dieser Vorgang ziemlich ähnlich vonstatten. Auch hier wird wieder ein Unterprogramm bei einem Reset angesprungen, das einige Bytes mit festen Werten vergleicht. Diese Routine ist im Speicher von $E1F0 - $E241 zu finden:

```
E1F0   LDX #$F5
E1F2   LDY #$FF
E1F4   STX $C3
E1F6   STY $C4
E1F8   LDA #$C3
E1FA   STA $02AA
E1FD   LDY #$02
E1FF   LDX #$7F
E201   JSR $02A2
E204   CMP $E2C4,Y
E207   BNE $E224
```

```
E209   DEY
E20A   BPL $E201
E20C   LDX #$F8
E20E   LDY #$FF
E210   STX $C3
E212   STY $C4
E214   LDY #$01
E216   LDX #$7F
E218   JSR $02A2
E21B   STA $0002,Y
E21E   DEY
E21F   BPL $E218
E221   JMP ($0002)
----------------------
E224   LDA #$40
E226   STA $FF00
E229   LDA #$24
E22B   LDY #$E2
E22D   STA $FFF8
E230   STY $FFF9
E233   LDX #$03
E235   LDA $E2C3,X
E238   STA $FFF4,X
E23B   DEX
E23C   BNE $E235
E23E   STX $FF00
E241   RTS
```

Zuerst wird ein Vektor ($C3/$C4) auf die Adresse $FFF5 gerichtet. Dann wird eine Routine angesprungen, die ein Byte von einer beliebigen Adresse in einer beliebigen Bank holt. Hier wird zuerst der Inhalt der Adresse $FFF7 in der Bank 1 (RAM-Bank 1 durchgehend) geholt. Dieses Byte wird dann mit dem Inhalt der Adresse $E2C6 verglichen. Sind diese beiden Bytes nicht identisch, so wird sofort zur Adresse $E224 gesprungen. Sind sie gleich, so werden die nächsten Bytes getestet. Insgesamt werden folgende Adressen verglichen:

```
$FFF5 mit $E2C4 : $43 (= "C")
```

```
$FFF6 mit $E2C5 : $42 (= "B")
$FFF7 mit $E2C6 : $4D (= "M")
```

Im Gegensatz zum 64er werden also nur drei Bytes getestet - was ja wohl auch genügt. Diese drei Bytes entsprechen dem String "CBM", der auch noch bei einer anderen Routine benutzt wird (siehe "Autostart mit Floppy).
Sind diese drei Bytes identisch, so werden die Inhalte der Adressen $FFF8 und SFFF9 geholt und in die Adressen $0002 und $0003 gespeichert. Über diesen Vektor wird dann ein indirekter Sprung ausgeführt.
Um einen Autostart durchzuführen, müssen also folgende Bytes folgendermaßen belegt sein:

```
$FFF5 : $43 ("C")
$FFF6 : $42 ("B")
$FFF7 : $4D ("M")
$FFF8 : Low-Byte gewünschte Sprungadresse
$FFF9 : High-Byte gewünschte Sprungadresse
```

Die Routine zu der bei Auftreten eines Fehlers gesprungen werden soll, muß im Speicher der Konfiguration, die bei Bank 15 gebildet wird, liegen. So geht es z. B. nicht, eine Routine bei $A000 im RAM der RAM-Bank 0 anzuspringen, da bei Bank 15 dort der BASIC-Interpreter liegt. Sie können aber eine Routine z. B. im Kassettenpuffer ($0B00 - $0BFF) anspringen, die dann die entsprechende Speicherkonfiguration einstellt.

Bevor wir einige Beispiele zur Verwendung des Autostarts zeigen, noch eine kurze Erklärung, wie die Routine weiterverfährt, wenn keine Übereinstimmung dieser Bytes vorliegt, denn im Gegensatz zum 64er wird hier nicht einfach zurückgesprungen.
Ab Adresse $E224 wird mit den ersten beiden Befehlen eine Speicherkonfiguration eingestellt, die der von Bank 15 gleich, nur daß statt RAM-Bank 0 die RAM-Bank 1 eingestellt ist (zur Erinnerung: RAM-Bank 15 stellt alle System-ROMs an). Anschließend wird folgendes in die Adressen $FFF5 - $FFF9 gespeichert:

```
$FFF5 : $43 ("C")
$FFF6 : $42 ("B")
$FFF7 : $4D ("M")
$FFF8 : $24
$FFF9 : $E2
```

Dadurch wird bei einem erneuten Reset auch ein Autostart ausgeführt, jedoch nur auf die normale Reset-Routine.

Schauen Sie sich nun die Bytes einmal an: Gehen Sie in den Monitor und tippen Sie "M 1FFF5" ein. Wie Sie sehen, sind die Adressen $FFF5 - $FFF9 (in diesem Falle besser $1FFF5 - $1FFF9) schon mit den oben erwähnten Werten belegt.
Geben Sie nun Ihre eigene Adresse für einen Autostart ein: Gehen Sie auf das Byte $24 und setzen Sie stattdessen $4D ein. Gehen Sie nun auf den Wert $E2 und tippen Sie dafür $FF. Dadurch wird statt zur Adresse $E224 bei einem auftretenden Reset zur Adresse $FF4D gesprungen. In dieser Adresse fängt die Routine C64mode an, d. h. der 128er verwandelt sich in einen 64er.
Probieren Sie es nun aus: Drücken Sie die Reset-Taste. Nach kurzer Zeit (der 80-Zeichen-Bildschirm fängt eventuell wild zu flackern an) befinden Sie sich im 64er.
Noch etwas nebenbei: Schalten Sie nun den Rechner kurz aus und dann wieder an. Wie Sie sehen, schaltet der Rechner in den 64er-Mode! Nein, keine Angst, er ist nicht kaputt. Beim 128er werden nur die verwendeten RAMs nicht so schnell gelöscht wie beim 64er. Dadurch bleiben die Werte von $FFF5 - $FFF9 noch im Rechner erhalten. Der Rechner findet sie beim Reset und führt - wie eben - die Routine c64mode durch. Schalten Sie den Rechner etwas länger aus und Sie kommen wieder anstandslos in den 128er-Modus. Diese lange Aufrechterhaltung des RAMs hat zwei Vorteile: Erstens beeinträchtigen kurze Stromausfälle nicht den Betrieb des Rechners und zweitens bleiben Ihnen Proramme nach kurzem Ausschalten (meistens) noch erhalten (Maschinenprogramme, sofern sie nicht in einem Bereich liegen, der bei Reset gelöscht wird sowieso, und BASIC-Programme können Sie mittels OLD wiederholen).

Natürlich können Sie auch BASIC-Programme bei einem Reset starten lassen. Sie müssen dabei den Vektor $FFF8/$FFF9 nur auf eine Routine legen, die den RUN-Befehl simuliert. Beim C-64 ging das durch folgende Assembler-Befehle:

```
JSR $A659   ;CHRGET auf Programmstart + CLR
JMP $A7AE   ;Interpreterschleife
```

Da wir jedoch noch über kein INTERN zum C-128 verfügten, konnten wir die entsprechenden Routinen beim C-128 nicht finden. Es geht jedoch auch anders: Da der C-128 auch über einen Tastaturpuffer verfügt, speichert man in diesen einfach den String "RUN + RETURN" ab und beendet das Programm. Daraufhin wird ein im Speicher befindliches BASIC-Programm gestartet. Als Assembler-Listing sähe das so aus:

```
       LDX #$02          ; Zähler
  loop LDA tabelle,X     ; Zeichen holen
       STA $034A,X       ; und in Tastaturpuffer
       DEX               ; Noch ein Zeichen?
       BPL loop          ; ja
       LDA #$03          ; Anzahl der Zeichen
       STA $D0           ; im Tastaturpuffer
       RTS

tabelle $52, $D5, $0D    ; ASCII-Werte für R, SHIFT + U
                         ; und RETURN
```

11. DER SPEICHER

11. Der Speicher

****

11.1. Nützliche Adressen des Commodore 128

Sicherlich, das neue BASIC des Commodore 128 läßt eigentlich kaum mehr Wünsche offen, was die eigene Programmierung anbelangt. Es läßt sich da allerdings noch eine ganze Menge mehr machen - ohne Maschinenprogrammierung. Schlüssel dazu ist die eingefleischten Programmierern wohlbekannte "Zeropage". Diese Bezeichnung stand ursprünglich für die ersten 256 Speicherstellen im Speicher des Rechners. In diesem Bereich befinden sich eine Vielzahl von Zeigern und Zwischenspeichern, die das Betriebssystem benötigt. Gerade deshalb sind sie jedoch auch für den BASIC-Programmierer interessant, denn mit einigen Speicheradressen läßt sich das Betriebssystem manipulieren, wie es mit normalen BASIC-Befehlen nicht oder nur sehr schwer möglich ist.
Das Betriebssystem des Commodore 128 ist gegenüber den ersten Commodore-Heimcomputern sehr viel komplexer geworden, und so reichte die nostalgische Größe der Zeropage von 256 Bytes bei weitem nicht mehr aus. Dieser Bereich umfaßt inzwischen mehrere Speicherseiten. Auf den folgenden Seiten finden Sie die interessantesten Speicherstellen - und was man mit Ihnen anstellen kann. Keine Angst: Durch das Verändern dieser Speicherstellen kann dem Computer kein Schaden zugefügt werden. Sollten Sie eigene "Ausflüge" in diesen Speicherbereich unternehmen, so "hängt" sich der Computer höchstens einmal "auf". Nach dem Wiedereinschalten ist ohnehin wieder alles beim Alten!

Neben den Speicheradressen finden Sie in Klammern die entsprechenden Adressen auf dem Commodore 64. Es dürfte also nicht sehr schwierig sein, alte C64-BASIC-Programme auf den 128-er Modus zu übertragen, da das C64-BASIC ohnehin kompatibel ist!

Interessante Speicherstellen und was man damit macht...

```
(dez.)          (Bezeichnung)
-------------------------------------------------------
```

45 - 46 (43-44) BASIC Start (Bank 0)

Diese Speicherstelle enthält im Lo/Hi-Byte-Format die Anfangsadresse, ab der BASIC-Programme im Speicher liegen (Bank 0). Die Abfrage:

```
PRINT PEEK(45)+256*PEEK(46)
```

ergibt diese Adresse. Ebenso läßt sich der BASIC-Start auch verschieben. Wollen Sie den BASIC-Start an eine andere Adresse legen, so gehen Sie folgendermaßen vor:

```
POKE 45,lo:POKE 46,hi:POKE (lo+256*hi)-1,0:NEW
```

Diese Befehlssequenz muß im Direktmode eingegeben werden. Lo und hi sind dabei Lo/Hi-Byte der neuen Anfangsadresse. Diese beiden Bytes ermitteln sich wie folgt:

```
HI=INT(add/256):LO=add-(256*HI)
```

Wurde der BASIC-Start verschoben, so ist das im Speicher befindliche Programm gelöscht (wie sich dies verhindern läßt, finden Sie an anderer Stelle in diesem Buch!). Beachten Sie bitte, daß der BASIC-Start normalerweise nicht nach unten verschoben werden darf, da dieser Speicherbereich für das Betriebssystem reserviert ist!

47 - 48 (45-46) VARIABLEN Start (Bank 1)

Analog zum oben beschriebenen BASIC-Start-Vektor zeigen diese beiden Bytes auf den Start der Variablenspeicherung (in Speicherbank 1). Auch dieser Zeiger kann auf oben genannte Art gelesen oder manipuliert werden.

59 - 60 (57-58) Augenblickliche BASIC-Zeilennummer

In diesen beiden Bytes ist die Zeilennummer der BASIC-Zeile gespeichert, die gerade abgearbeitet wird. So ist das Auslesen dieser Adressen auch nur im Programm-Mode sinnvoll:

```
10 PRINT "Im Moment wird gerade Zeile Nr. ";PEEK(59)+
   256*PEEK(60);" abgearbeitet!"
```

65 - 66 (63-64) Augenblickliche DATA-Zeilennummer

Sofern Sie in Ihrem Programm die Befehle READ und DATA verwenden, könnte dieser Zeiger sehr interessant werden. Er beinhaltet nämlich die Zeilennummer der Zeile, aus der gerade das aktuelle DATA-Element geholt wird. Das folgende Programm demonstriert dies:

```
 10 READ A: IF A=-1 THEN END
 20 PRINT"DAS ELEMENT ";A;" STAMMT AUS ZEILE";
 30 PRINT PEEK(65)+256*PEEK(66);"!"
 40 GOTO 10
 50 DATA 3,6,4,8,4,6,2
 57 DATA 33,6,4,2,4,2,4
 99 REM TESTPROGRAMM
167 DATA 3,7,4,9,6,0,0
190 DATA 5,7,5,-1
```

Auch zur Fehlererkennung ist diese Adresse nützlich. Angenommen, Ihr Programm erwartet Zahlen als DATAs. Unglücklicherweise hat sich jedoch ein String dazwischengemogelt. Dies wird erkannt und mit Hilfe dieses Zeigers kann komfortablerweise auch gleich die Zeilennummer geliefert werden, in der sich der Fehler befindet:

```
10 READ A$: IF ASC(A$)<50 OR ASC(A$)>60 THEN GOTO 1000
20 A=VAL(A$):IF A=-1 THEN END
30 PRINT A,:GOTO 10
40 DATA 4,3,7,54,3,5,2,444
```

```
  50 DATA 3,5656,a,3,d,4,2,2,2:REM FEHLERHAFTE ZEILE
  60 DATA 4,6,3,5,6
  70 :
1000 REM FEHLERMELDUNG
1010 PRINT"IN ZEILE ";PEEK(65)+256*PEEK(66);
1020 PRINT"FEHLER GEORTET WORDEN. ES DUERFEN NUR ZAHLEN"
1030 PRINT"VERWENDET WERDEN !"
1040 PRINT"AENDERN SIE DAS !":END
```

208 (198) Zeiger im Tastaturpuffer

Dieser Zeiger enthält die Anzahl der vom Computer zwischengespeicherten Tastendrücke. Einzelheiten darüber erfahren Sie im Kapitel "Zeilen einfügen in ein Programm".

```
POKE 208,0:GETKEY A$
```

verhindert, daß vor der eigentlichen Abfrage gedrückte Tasten übernommen werden.

213 (203) Tastaturabfrage

In dieser Adresse ist der Wert der aktuell gedrückten Taste abgelegt. Ist keine Taste gedrückt, so enthält sie den Wert 88. Mehr darüber finden Sie im Kapitel über Tastaturabfrage.

```
10 PRINT CHR$(147);
20 PRINT PEEK(213):GOTO 10
```

gibt den Wert der gerade gedrückten Taste aus.

215 (---) 40/80-Zeichen-Flag

Diese Adresse gibt Auskunft darüber, welcher Bildschirm eingeschaltet ist. Bit 7 wird im 80-Zeichen-Modus gesetzt:

```
10 A=PEEK(215)
```

```
20 IF A=0 THEN PRINT"40-Zeichen-Schirm":ELSE
   PRINT "80-Zeichen-Schirm"
```

oder:

```
10 PRINT"Durch diese Programmiertechnik wird ";
20 IF PEEK(215)=0 THEN PRINT
30 PRINT"gewaehrleistet, dass die ausgegebenen"
40 :
50 PRINT"PRINT-Zeilenlaengen immer der gerade";
60 IF PEEK(215)=0 THEN PRINT
70 PRINT"aktuellen Zeilenlaenge angepasst werden!"
```

Die folgenden Adressen beziehen sich auf den 40- bzw. 80-Zeichen-Bildschirm. Diese Speicherstellen werden beim Umschalten auf den anderen Bildschirm ausgelagert. So ist hier immer der Wert für den aktuellen Bildschirm zu finden!

---

241 (646) Aktuelle Zeichenfarbe

Es stehen insgesamt 16 Farben zur Verfügung, mit denen Zeichen auf den Bildschirm gebracht werden können. Normalerweise dienen Farb-Steuerzeichen dazu, diese Farben einzuschalten. Dies ist mitunter im Listing sehr unübersichtlich und auch unpraktisch. Einfacher geht es oft mit Hilfe dieser Speicherstelle. Sie enthält den Wert der aktuellen Zeichenfarbe (0-15) und kann sehr einfach manipuliert werden. Hier ein Beispiel:

```
10 A=INT(RND(1)*16):REM Zufallszahl 0-15
20 POKE 241,A:REM Zeichenfarbe setzen
30 PRINT "*";
40 GOTO 10
```

Versuchen Sie das einmal mit Farb-Steuerzeichen!

Hier noch einmal eine kurze Zusammenstellung der Farbkombinationen:

```
-------------------------------------------------------
 0 : schwarz        1 : weiß          2 : rot
 3 : türkis         4 : violett       5 : grün
 6 : blau           7 : gelb          8 : hellrot
 9 : braun         10 : hellrot      11 : grau 1
12 : grau 2        13 : hellgrün     14 : hellblau
15 : grau 3

    (gilt nur für 40-Zeichen-Schirm!)
-------------------------------------------------------
 0 : schwarz        1 : grau 2        2 : blau
 3 : hellblau       4 : grün          5 : hellgrün
 6 : grau 1         7 : türkis        8 : rot
 9 : hellrot       10 : dunkelviolett 11 : violett
12 : braun         13 : gelb         14 : grau 3
15 : weiß

    (gilt nur für 80-Zeichen-Schirm!)
-------------------------------------------------------
```

Falls Sie mit einem Grünmonitor arbeiten, haben Sie natürlich nichts von diesen vielen bunten Farben. Aber die Zeichen werden je nach Farbe mit verschieden großer Helligkeit auf den Grünmonitor gebracht. Sie sollten beachten, daß es trotz 16 Farben lediglich fünf Helligkeitsstufen gibt. Wollen Sie also auch auf einem Grünmonitor Farben (der Helligkeit gemäß) einsetzen, sorgen Sie dafür, daß Sie Farben verschiedener Helligkeitsstufen wählen. Hier die Stufen:

1. schwarz
2. rot, blau, braun, grau1
3. violett, grün, orange, hellrot, grau2, hellblau
4. türkis, gelb, hellgrün, grau3
5. weiß

Und nun noch etwas ganz Spezielles für die Benutzer des 80-Zeichen-Schirmes: Hier haben die übrigen Bits 4-7 weitere Funktionen bei der Zeichendarstellung:

```
Bit 4 : blinken
Bit 5 : unterstreichen
Bit 6 : revers
Bit 7 : 2. Zeichensatz
```

Diese Funktionen können zusätzlich benutzt werden. Bit 4 und 5 haben jedoch nur Einfluß auf eine unmittelbar folgende PRINT-Anweisung.

```
POKE 241,1+2^4:PRINT "TEST"
```

bringt ein graues, blinkendes "TEST" auf den Bildschirm.

```
POKE 241,2+2^5+2^7:PRINT "HALLO"
```

printet ein blaues, unterstrichenes "HALLO". Die Funktionen können also auch gemischt werden!

Die Funktionen "revers" und "2. Zeichensatz" sind Dauerfunktionen; sie beziehen sich auf ALLE folgenden PRINT-Anweisungen und den Direktmode. Ein so eingeschalteter RVS-Mode kann nicht durch CTRL+9 aufgehoben werden!

243 (199) RVS-Flag

Analog zu Adresse 241 bestimmt diese Adresse die Darstellungsart eines auf den Bildschirm auszugebenden Zeichens: revers oder normal (auch hierfür gibt es ein Steuerzeichen). 0=normal, 1=revers:

```
10 POKE 243,0:PRINT "NORMAL und ...";
20 POKE 243,1:PRINT "...REVERS !"
```

244 (212) Quote-Mode-Flag

Nachdem wir Ihnen in den obigen beiden Beispielen so schön gezeigt haben, wie Sie Steuerzeichen überflüssig machen können, geht es ihnen nun höchstselbst an den Kragen. Quote-Mode bedeutet nichts weiter als "Anführungszeichen-Mode". Steuerzeichen wirken nur innerhalb von Anführungszeichen. Schaltet man nun mit dieser Adresse in den Quote-Mode (=1), so schaltet eine nachfolgende PRINT-Anweisung diesen wieder ab anstatt ein. Das Ergebnis: Eventuell vorkommende Steuerzeichen in dieser PRINT-Anweisung werden nicht ausgeführt, sondern auf den Bilschirm ausgegeben. Sehen Sie selbst:

```
POKE 244,1:PRINT"(3xcrsrdwn)(rvson)Hallo !"
```

245 (657) C=/SHIFT Sperr-Flag

Sicherlich kennen Sie die Möglichkeit, durch gleichzeitiges Drücken der Tasten C= und SHIFT den aktuellen Zeichensatz umzuschalten. Dies läßt sich auch während eines laufenden Programms tun. Es kann zwar den Programmlauf nicht beeinträchtigen, die Optik hingegen leidet oft. Perfektionieren Sie doch Ihre Programme, indem Sie diese Umschaltmöglichkeit einfach unterbinden:

```
POKE 245,64   (verriegelt)
POKE 245,0    (normal)
```

248 (---) Scrolling Sperr-Flag

Sobald Sie die unterste Bildschirmzeile erreicht haben und die nächstfolgende beschreiben möchten, wird normalerweise der Bildschirminhalt um eine Zeile nach oben geschoben, um Platz zu schaffen. Dieser Effekt läßt sich jedoch abschalten:

```
POKE 248,128 (Scrolling aus)
POKE 248,0   (normal)
```

249 (---) Beep-Ton Sperr-Flag

Sie merken schon, es kann eine ganze Menge gesperrt werden. Hier ist Ihnen eine Möglichkeit an die Hand gegeben, den hauseigenen Piep-Ton zu verriegeln. Normalerweise erklingt dieser durch:

```
   PRINT CHR$(7); (Programm-Mode)
oder
      CTRL+G (Direkt-Mode)
```

Wen das stört, versuche:

```
   POKE 249,128  (verriegelt)
   POKE 249,0    (weckt den Quälgeist wieder)
```

Übrigens: SYS 51602 läßt den Ton erklingen (nur bei Monitoren mit eingebautem Audio-Teil!).
Mit dem Ton läßt sich aber mehr machen als eine einfache Signalwirkung:

```
   FOR X=0 TO 100:SYS 51602:NEXT X
```

läßt einen mehr oder weniger langen Ton erklingen. Variieren Sie doch einfach die Werte für X Ihren Wünschen entsprechend!

Alle nun folgenden Adressen gelten wieder allgemein und sind nicht an den aktuellen Bildschirm gebunden. Sie liegen auch nicht mehr in der eigentlichen "Zeropage".

---

842 - 852 (631-640) Tastaturpuffer

Diese zehn Bytes bilden den Zwischenspeicher des Rechners für Tastatureingaben. Er kann allerdings programmtechnisch

äußerst sinnvoll genutzt werden. Die möglichen Anwendungen sprengen den Rahmen dieser Kurzbeschreibung. Sie finden daher ausführliche Einzelheiten im Zusammenhang mit interessanten Programmen in einem eigenen Kapitel über den Tastaturpuffer in diesem Buch!

```
10 POKE 842,ASC("L"):POKE 843,ASC("I")+128
20 POKE 844,13
30 POKE 208,3
40 END
```

1024 - 2023 (dito) 40-Zeichen-Bildschirmspeicher

Diese 1000 Bytes enthalten den Inhalt des 40-Zeichen-Bildschirmes. Arbeiten Sie ausschließlich mit dem 80-Zeichen-Schirm, so ist dieser Bereich ungenutzt und kann für eigene Maschinenprogramme genutzt werden. Ansonsten gilt:

```
POKE 1024+X+40*Y,Z
```

X=Spalte
Y=Zeile
Z=Zeichencode

bringt ein beliebiges Zeichen auf den 40-Zeichen-Bildschirm.

2592 (649) Maximale Länge des Tastaturpuffers

Die Länge des Tastaturpuffers wird von dieser Adresse bestimmt. Einzelheiten über die Wirkungsweise des eigentlichen Puffers finden Sie im Kapitel "System der 1000 Möglichkeiten: Zeilen einfügen".

```
POKE 2592,0  (schaltet den Puffer aus, keine Speicherung mehr von Tastendrücken!)
POKE 2592,10 (normal)
```

Die Größe des Tastaturpuffers sollte 10 normalerweise nicht überschreiten, da dies der regulär maximal zur Verfügung stehende Speicherplatz ist!

2593 (---) CTRL-S Flag

Probieren Sie es einmal aus: Ein im Ablauf befindliches Programm stoppt sofort, wenn Sie gleichzeitig die CTRL-Taste und "S" drücken. Wieder weiter geht's mit jeder anderen Taste.
Diese Funktion besitzen eine ganze Reihe "großer" Rechner. Sie kann außerordentlich nützlich sein, will man das Programm kurz unterbrechen, um sich etwas genauer anzusehen oder kurz zu frühstücken.

Man kann sich diese Adresse aber in einer ganz anderen Weise nützlich machen. Man simuliert vom laufenden Programm aus die Betätigung CTRL+S. Die Folge: Das Programm stoppt und wartet auf einen Tastendruck:

```
10 PRINT "PRESS ANY KEY !"
20 POKE 2593,1
30 PRINT "OK"
```

Man spart sich so viel Ärger mit GET oder GETKEY.

2594 (650) REPEAT-Flag

Alle C64-Umsteiger werden es wohl früher oder später zu spüren bekommen haben: Der C128 besitzt von vornherein eine REPEAT-Funktion für alle Tasten, d.h. wird eine Taste längere Zeit gedrückt, so wird das entsprechende Zeichen fortlaufend ausgegeben. Dabei sind in dieser Beziehung C128 und C64 völlig gleich geblieben: Bei beiden Rechnern gibt es eine Speicherstelle, die über die REPEAT-Funktion entscheidet. Lediglich der Wert dieser Speicherstelle ist unterschiedlich, die Belegung dagegen dieselbe geblieben:

```
POKE 2594,0     (Tastenwiederholung aus; wie C64)
POKE 2594,64    (Verriegelung: Absolut keine Taste be-
                 sitzt mehr Wiederholungsfunktion)
POKE 2594,128   (Tastenwiederholung für alle Tasten ein;
                 normal)
```

2595 (651) REPEAT-Verzögerung

Erst, wenn eine Taste lang genug gedrückt wird, beginnt der Computer mit der fortlaufenden Ausgabe dieses Zeichens. Wie lange diese Verzögerung dauern soll, kann durch diese Adresse bestimmt werden. So ist es zum Beispiel möglich, einen großen "Sicherheitsabstand" zu wählen, wenn die REPEAT-Funktion nur zeitweise benutzt werden soll, oder aber gar keine Verzögerung zu wählen. Dies ist beispielsweise sinnvoll, wenn (z.B. mittels GET) eine Steuerung realisiert werden soll.

```
POKE 2595,X
```

X=Dauer der Verzögerung

2598 (---) VIC Cursor-Mode

Diese Adresse steuert das Verhalten des Cursors. Es handelt sich hierbei jedoch nur um den vom VIC gesteuerten Cursor, funktioniert also auch nur bei eingeschaltetem 40-Zeichen-Mode.

```
Bit 6 : 1=solid
        0=blink

POKE 2598,0  (Cursor blinkt, normal)
POKE 2598,64 (Cursor steht fest)
```

2599 (204) VIC Cursor Ein-/Aus-Flag

Auch der Zuständigkeitsbereich dieser Adresse bezieht sich nur auf den 40-Zeichen-Schirm. Mit dieser Adresse kann der Cursor im Programm eingeschaltet werden. Damit läßt sich eine ganze Menge machen!

```
Bit 0 : 1=Cursor aus
        0=Cursor ein

10 POKE 2599,0:REM CURSOR EIN
20 PRINT "HALLO!";
30 FOR T=1 TO 5000: NEXT T
40 PRINT"DAS WAR'S!"
50 END
```

...demonstriert den Programm-Cursor. Eine sinnvollere Anwendung ist das folgende Programm. Sicher werden Sie den neuen Befehl "GETKEY" kennen. Er wartet auf einen Tastendruck und speichert das Zeichen ab. Anders als beim INPUT-Befehl erscheint jedoch kein blinkender Cursor, der den Anwender auf die Eingabe aufmerksam macht. Hier kann Abhilfe geschaffen werden:

```
10 POKE 2599,0
20 GETKEY A$
30 (...)
```

Dasselbe funktioniert natürlich auch bei GET.

2603 (----) VDC Cursor-Mode

Analog zu Adresse 2598 kann hier der Cursor des 80-Zeichen-Schirms beeinflußt werden. Dabei haben die Bits dieser Adresse folgende Bedeutung:

```
Bit 0      : 1=solid
Bit 2-3    : Darstellung in Pixelzeilen
Bit 4-5    : Cursor aus
Bit 6      : 1=blink

POKE 2603,2^0+2^1+2^2
```

schaltet beispielsweise einen sogenannten "Underline"-Cursor an, wie er bei vielen Großrechnern anzutreffen ist. Die Dicke des Cursors ist dabei von den Bits 2 und 3 abhängig.

2604 (----) VIC Pointer auf BS-RAM/Zeichengenerator

Der Inhalt dieser Adresse wird automatisch in Adresse 53272 des VIC übertragen. Dort entscheidet der Inhalt über Startadressen des Bildschirm-RAMs und des Zeichengenerators. Nähere Informationen über die möglichen Startkombinationen und Anwendungsbeispiele finden Sie im Kapitel "Der 40-Zeichen-Charactergenerator").

2606 (648) VDC Hi-Byte des Bildschirm-RAMs

Diese Adresse ist wieder zuständig für den 80-Zeichen-Schirm. Dort kann der Bildschirmspeicher über die Register 12 und 13 des VDC verschoben werden. Das High-Byte der Anfangsadresse des Bildschirm-RAMs muß jedoch auch hier abgespeichert werden, um ordnungsgemäß funktionieren zu können (siehe "Verschieben von BS- und Attribut-RAM").

2607 (----) VDC Hi-Byte des Farbspeichers (Attribut-RAM)

Analog zur vorangegangenen Adresse ist hier das High-Byte des 80-Zeichen-Attribut-RAMs abgelegt, da im 80-Zeichen-Mode der Farbspeicher verschoben werden kann.

2619 (648) VIC Hi-Byte des Bildschirm-RAMs

Diese Adresse hat dieselbe Aufgabe wie Adresse 2606 beim VDC. In ihr ist das Hi-Byte der Anfangsadresse des 40-Zeichen-Bildschirm-RAMs abgelegt.

2816 - 3327 (828-1019) Kassettenpuffer

Dieser relativ große Speicherbereich wird nur dann vom System beansprucht, wenn mit der Datasette gearbeitet wird. Benutzen Sie ohnehin ein Floppy Disk Laufwerk, so ist dieser Bereich ungenutzt. Er dürfte wohl selbst für längere Maschinenroutinen Platz genug bieten!

4096 - 4105 (----) Länge der Funktionstastenbelegung

Jedes dieser zehn Bytes ist für eine der zehn Funktionstasten reserviert (F1-8, HELP und SHIFT+RUN-STOP). In den Bytes ist die Länge der Funktionstastenbelegung hinterlegt. Wollen Sie die Belegung einer Funktionstaste ausschalten (z.B. wenn die Taste in einem eigenen Programm abgefragt werden soll), so genügt es, das entsprechende Byte auf Null zu setzen!

```
POKE 4096,0  (löscht Belegung von F1)
```

4624 - 4625 (55-56) Zeiger auf BASIC-Ende

In diesen beiden Bytes ist die größtmögliche Adresse für BASIC-Programme (in Bank 0) gespeichert:

```
PRINT PEEK(4624)+256*PEEK(4625)
```

Die Größe des für BASIC-Programme zur Verfügung stehenden Speichers läßt sich leicht ermitteln:

```
PRINT (PEEK(4624)+256*PEEK(4625))-(PEEK(45)+256*PEEK(46))
```

Falls Sie eigene Maschinenprogramme an das Ende der Speicherbank 0 (BASIC-Speicher) legen möchten, können Sie mit Hilfe dieser beiden Adressen den BASIC-Speicher so einschränken, daß Ihre Maschinenprogramme vor dem überschreiben geschützt werden!

11.2. Sprungtabellen

Gute Kenntnisse des Betriebssystems sind wichtig für jeden Maschinensprache-Programmierer, da er so kürzere Programme bei geringerem Zeitaufwand programmieren kann. Statt eigene Routinen (z. B. Rechenroutinen) zu entwerfen, greift er einfach auf fertige Routinen zurück. Wir wollen hier nun nicht ein zweites INTERN schreiben, sondern nur einige Routinen herauspicken und deren Gebrauch zeigen.

In den BASIC-Interpreter und in das Betriebssystem des C-128 wurden mehrere Sprungtabellen mit leistungsfähigen und daher oft nutzbaren Routinen eingebunden.
Kommen wir gleich zu der bekanntesten und wohl auch wichtigsten:

11.2.1. Kernal

Wie in jedem Commodore-Rechner befindet sich auch im C-128 am unteren Ende des ROMs eine Kernal-Sprungtabelle, die problemlosen Zugriff auf wichtige Routinen des Kernals und eine schnelle Anpassung an andere Rechnertypen (natürlich nur die von Commodore) ermöglicht. Beim 128 wurde diese Sprungtabelle im Vergleich zum C-64 oder VC-20 um einige Routinen erweitert. Dieser Teil der Tabelle, von $FF4D - $FF7F, wird hier nun vorgestellt.
Zuerst einmal ein paar einleitende Worte zu dieser Tabelle. Die Routinen der alten Tabellen waren bei allen Rechner gleich gut zu gebrauchen. Hier gibt es nun einige Routinen, die speziell für den C-128 entworfen worden sind. So gibt es z. B. eine Routine c64mode, die den 128er in einen 64er umstellt. Welchen Sinn hätte diese Routine bei einem neuen Commodore-Rechner (AMIGA mit C-64 Modus?). Des weiteren gibt es Routinen, mit denen man auf eine andere Speicher-Bank zugreifen kann. Mehrere Banks könnte es zwar auch bei einem anderen Rechner geben (obwohl der 68000 Prozessor immerhin 16 MByte adressieren kann), jedoch hätte dann die Bank-Nummer, die man diesen Routinen übergeben muß, eine andere Konfiguration des Speichers zur Folge als beim 128er. Mehrere Routinen lassen sich jedoch auf einem anderen Rechner genauso gut einsetzen wie hier.

Interessant ist auch, daß Commodore ein Byte zwischen den beiden Tabellen (&FF80) mit Null belegt hat. Den Systeminformationen war zu entnehmen, daß das die Nummer der Kernal-Tabelle ist. Es wird also auch von Commodore aus ein Unterschied getroffen.
Kernal-Tabellen und wies so daraufhin.Zusätzlich zu der Sprungtabelle wurde auch noch eine neue Adresse direkt vor den Adressen für NMI, Reset und IRQ in den Speicher eingefügt: $FFF8/$FFF9 weist auf Adresse $E224 und nennt sich c128mode.

Doch lassen Sie sich nun in die Geheimnisse der neuen Sprungtabelle einweisen.

Kernal-Adresse: $FF4D (65357)
Name: c64mode
Tatsächliche Sprungadresse: $E24B (57931)
Funktion: Umgestaltung des 128er als 64er

Der Name dieser Funktion dürfte wohl selbsterklärend sein.
Nähere Erklärungen finden Sie im Kapitel 12: "Wechsel des Betriebssystems".

Kernal-Adresse: $FF50 (65360)
Name: dma-call
Tatsächliche Sprungadresse: $F7A5 (63397)
Funktion: Zugriff auf den dma-controller

Mit dieser Routine kann man mit dem dma-controller in Kontakt treten. Da sich dieses IC jedoch erst bei externen RAM-Erweiterungen entfaltet, ist diese Routine auch erst bei späteren Versionen des 128er mit mehr als 128K RAM interessant.

Kernal-Adresse: $FF53 (65363)
Name: boot-call
Tatsächliche Sprungadresse: $F890 (63632)
Funktion: Lade & Starte Programm von Disk

Mit dieser Routine kann man ein Programm auf Disk laden und sofort ausführen lassen. Da diese Routine auch bei einem Reset angesprungen wird, ist es möglich, Programme sofort nach dem Einschalten zu starten. Dadurch werden diese Programme natürlich erheblich benutzerfreundlicher, da sich ein Benutzer keine umständlichen LOAD-Befehle merken muß.
Nähere Erklärungen zu dieser Routine finden Sie im Kapitel 10.1: "Booting - Autostart mit Floppy".

Kernal-Adresse: $FF56 (65366)
Name: phoenix
Tatsächliche Sprungadresse: F867 (63591)
Funktion: Starten von Funktions-ROMs

Mit dieser Routine kann man eventuell angeschlossene Funktions-ROMs zum Starten bringen. Nach dem Abarbeiten von phoenix wird gleich weiter zu call-boot gesprungen.
Diese Routine wird auch bei einem Reset ausgeführt.

Kernal-Adresse: $FF59 (65369)
Name: lkupla
Tatsächliche Sprungadresse: $F79D (63389)
Funktion: Eintrag zu Dateinummer holen

Bei Aufruf dieser Routine gibt man im Akku die Dateinummer an und erhält die Einträge dazu zurück:

```
Akku       : Dateinummer
X-Register : Geräteadresse
Y-Register : Sekundäradresse
```

Wurde die Dateinummer nicht gefunden, so wird mit gesetztem, sonst mit gelöschtem Carry-Flag zurückgesprungen.
Diese Routine benutzt eine andere sehr interessante Routine: $F202. Diese Routine schaut nach, ob eine gegebene Dateinummer (im X-Register) schon vorhanden ist. Existiert sie schon, so wird mit gelöschtem Zero-Flag zurückgesprungen, sonst mit gesetztem.

Kernal-Adresse: $FF5C (65372)
Name: lkupsa
Tatsächliche Sprungadresse: $F786 (63366)
Funktion: Eintrag zu Sekundäradresse holen

Diese Funktion entspricht der vorherigen, nur wird der Eintrag zu einer, im Y-Register übergebenen, Sekundäradresse geholt. Es wird nicht nachgeschaut, ob mehrere Files mit der gesuchten Sekundäradresse existieren - die Routine gibt immer die Einträge zu dem File zurück, daß zuerst mit dieser Sekundäradresse eröffnet wurde.
Die drei Register enthalten nach dem Rücksprung die gleichen Werte wie bei der vorherigen Routine:

```
Akku       : Dateinummer
X-Register : Geräteadresse
Y-Register : Sekundäradresse
```

Auch die Flags werden entsprechend gesetzt.

Kernal-Adresse: $FF5F (65375)
Name: swapper
Tatsächliche Sprungadresse: $C02A (49194)
Funktion: Umschalten zwischen 80/40-Zeichen-Bildschirm

Bei Ansprung dieser Routine wird der Bildschirm gewechselt. War vorher der 40-Zeichen-Bildschirm in Gebrauch, so erscheinen nun Ihre sagenhaften Eingaben auf dem 80-Zeichen-Bildschirm.
Diese Routine wollen wir uns einmal "reinziehen":
Von der Adresse $FF5F wird zu $C02A gesprungen, von dort aus direkt weiter zu $CD2E (warum einfach, wenn es auch kompliziert geht?).
Bei $CD2E wird schließlich folgendes ausgeführt:

```
CD2E   LDX #$1A
CD30   LDY $0A40,X
CD33   LDA $E0,X
CD35   STA $0A40,X
```

```
CD38   TYA
CD39   STA $E0,X
CD3B   DEX
CD3C   BPL $CD30
CD3E   LDX #$0D
CD40   LDY $0A60,X
CD43   LDA $0354,X
CD46   STA $0A60,X
CD49   TYA
CD4A   STA $0354,X
CD4D   DEX
CD4E   BPL $CD40
CD50   LDA $D7
CD52   EOR #$80
CD54   STA $D7
CD56   RTS
```

Im ersten Abschnitt (bis $CD3D) werden die Speicherblöcke $E0 - $FA und $0A40 - $0A5A ausgetauscht. Daraus schließen Sie natürlich gleich, daß bei $0A40 - $0A5A immer die Bildschirm-Werte (Cursorposition, Insert-Mode-Flag, Quote-Mode-Flag, etc., etc.) des gerade nicht benutzten Bildschirms abgespeichert werden. Wenn Sie dann zurückschalten, geht's genau so weiter, wie der Schirm verlassen wurde. Probieren Sie's aus:
Gehen Sie in den Quote-Mode, indem Sie ein Hochkomma (oder auch Gänsefüßchen) eingeben. Drücken Sie nun ein paarmal die Cursortasten. Gehen Sie nun mit ESC + X (schön langsam hintereinander drücken) in den anderen Bildschirm und geben Sie irgendetwas ein. Wechseln Sie nun wieder zurück und drücken Sie ein paarmal die Cursortasten. Wie Sie sehen, befinden Sie sich immer noch im Quote-Mode.

Noch ein Test: Wechseln Sie noch einmal den Bildschirm und geben Sie ein:

```
POKE DEC("0A54"),0
```

Gehen Sie nun mit dem Hochkomma zurück in den Bildschirm und

drücken Sie eine Cursortaste - nun wird kein Steuerzeichen ausgegeben, sondern der Cursor bewegt sich. Warum? Nun, durch den Poke wurde einfach das Quote-Mode-Flag auf Null (bei eingeschaltetem Quote-Mode 1) gesetzt. Beim Zurückschalten wurde dieses Register kopiert und der Quote-Mode war weg. Sie können also durch POKEn vorausbestimmen, wie der gerade nicht benutzte Bildschirm nach dem Zurückschalten aussieht. Schauen Sie sich dazu auch einmal die Adressen in der Zeropage-Beschreibung an ($E0 - $FA).
Da Formeln immer ankommen, nun noch eine zur Umrechnung einer Adresse des stillgelegten Bildschirms in die entsprechende Adresse des gerade benuzten:

Neue Adresse = Alte Adresse + 2400 (toll, ne?)

Doch kommen wir zu der Swapper-Routine zurück. Die nächsten acht Befehle tauschen zwei weitere Blöcke aus, nämlich

```
a) $0354 - $035E (Tabulator-Stops) mit $0A60 - $0A6A
b) $035F - $0361 (???) mit $0A6B - $A6D
```

In den Adressen $CD50 - $CD55 wird schließlich der Bildschirm umgeschaltet. Dazu wird das Bit 7 von $D7 invertiert, d. h. wenn es gerade gesetzt war (80 Zeichen) wird es gelöscht (40 Zeichen) und umgekehrt.
Damit ist diese Routine auch schon beendet.

Kernal-Adresse: $FF62 (65378)
Name: dlchr
Tatsächliche Sprungadresse: $C027 (49191)
Funktion: Initialisieren des VDC Zeichengenerators

Diese Routine kopiert den Zeichengenerator des VIC (zuständig für 40 Zeichen) in den Zeichengenerator des VDC (zuständig für 80 Zeichen). Der Zeichengenerator des VDC wird dabei jedoch doppelt so groß (8K), da zwischen jedes Zeichen acht Leerbytes kopiert werden. Den Sinn des ganzen erfahren Sie im Kapitel 1.

Kernal-Adresse: $FF68 (65384)
Name: setbnk
Tatsächliche Sprungadresse: $F73F (63295)
Funktion: Bestimmung der Bank für SAVE/LOAD/VERIFY

Diese Routine besteht aus nur drei Befehlen:

```
$F73F  STA $C6 ; Banknummer, in der das Programm steht
$F741  STX $C7 ; Banknummer, in der der Filename steht
$F743  RTS
```

Die Adresse des Filenamens geht aus dem Vektor $BB / $BC hervor.

Kernal-Adresse: $FF6B (65387)
Name: getcfg
Tatsächliche Sprungadresse: $F7EC (63468)
Funktion: Holen des CR-Bytes zur Bank-Nummer

Bevor Sie diese Routine anspringen, müssen Sie das X-Register mit der Nummer der gewünschten Bank laden (0-15). Sie erhalten dann im Akku ein Byte zurück. Dieses Byte entspricht der Belegung des configuration register der MMU, damit die gewünschte Bank ausgewählt wird.
Sie müssen es also bloß noch in der Adresse $D500 (bzw. $FF00) speichern.
Welche Speicherkonfiguration die einzelnen Bank-Nummern zur Folge haben, erfahren Sie in dem Kapitel über die MMU.

Kernal-Adresse: $FF6E (65390)
Name: jsrfar
Tatsächliche Sprungadresse: $02CD (717)
Funktion: Starten eines Unterprogramms in beliebiger Bank

Mit dieser Funktion kann man ein Unterprogramm in einer beliebigen Bank aufrufen. Nach der Abarbeitung kehrt der Computer zu der aufrufenden Bank zurück.
Vor dem Aufruf (der mit JSR zu erfolgen hat) müssen Sie folgende Speicherstellen und Register mit den entsprechenden Inhalten laden:

```
$0002 : Nummer der gewünschten Bank
$0003 : High-Byte der gewünschten Adresse
$0004 : Low-Byte der gewünschten Adresse
$0005 : Gewünschtes Status-Register (damit kann z. B.
        der IRQ vor dem Ansprung ausgeschaltet werden,
        auf Dezimalmodus umgeschaltet werden etc.)
$0006 : Gewünschter Akku
$0007 : Gewünschtes X-Register
$0008 : Gewünschtes Y-Register
$02DE : Nummer der Bank, von der aus aufgerufen wird
```

Nach dem Rücksprung enthalten folgende Adressen die neuen Registerinhalte:

```
$0005 : Status
$0006 : Akku
$0007 : X-Register
$0008 : Y-Register
$0009 : Stapel-Zeiger
```

Was Sie mit diesen Werten dann machen, ist natürlich Ihre Sache. So können Sie Speicherstelle $0006 auch in das X-Register laden und so einen Transfer-Befehl sparen.

Kernal-Adresse: $FF71 (65393)
Name: jmpfar
Tatsächliche Sprungadresse: $02E3 (739)
Funktion: Springen zu einer Bank

Das ist nun das Gegenstück zu dem vorherigen Befehl. Hier wird ein JMP zu einer beliebigen Bank ausgeführt. Da diese Routine auch von jsrfar benutzt wird, geschieht der Aufruf sehr ähnlich:

```
$0002 : Nummer der gewünschten Bank
$0003 : High-Byte der gewünschten Adresse
$0004 : Low-Byte der gewünschten Adresse
$0005 : Gewünschtes Status-Register
$0006 : Gewünschter Akku
```

```
$0007 : Gewünschtes X-Register
$0008 : Gewünschtes Y-Register
```

Im Gegensatz zu jsrfar muß hier der Aufruf mit JMP erfolgen.

Kernal-Adresse: $FF74 (65369)
Name: indfet
Tatsächliche Sprungadresse: $F7D0 (63440)
Funktion: Holen eines Bytes von beliebiger Bank

Mit dieser Routine kann man den Inhalt einer beliebigen Adresse einer beliebigen Bank holen.
Rufen Sie die Routine folgendermaßen auf:

```
$02B9 : Low-Byte des Vektors, der auf die gewünschte
        Speicherstelle zeigt. Der Vektor muß natürlich
        vorher initialisiert worden sein (er muß die
        gewünschte Adresse zerlegt in Low- und High-
        Byte enthalten). Dieser Vektor muß in der
        Zeropage liegen. Freie Bytes finden Sie im
        Anhang in dem Zeropage-Listing.
Akku  : Byte, das in die gewünschte Adresse
        geschrieben werden soll.
X-Reg.: Nummer der gewünschten Bank
Y-Reg.: Offset, der ggf. zu der Adresse addiert wird
        (normalerweise Null, wenn nicht gerade eine
        Tabelle abgefragt werden soll).
```

Kernal-Adresse: $FF77 (65399)
Name: indsta
Tatsächliche Sprungadresse: $F7DA (63450)
Funktion: Speicherung eines Bytes in beliebiger Bank

Hier kommt nun das Gegenstück zu indfet.
Um ein Byte abzuspeichern, müssen folgende Adressen / Register gesetzt worden sein:

```
Akku  : Low-Byte des Vektors (der wie bei indfet auf
        die gewünschte Adresse zeigen muß).
X-Reg.: Nummer der gewünschten Bank
Y-Reg.: Offset
```

Kernal-Adresse: $FF7A (65402)
Name: indcmp
Tatsächliche Sprungadresse: $F7E3 (63459)
Funktion: Vergleichen von Akku und Byte in beliebiger Bank

Diese Funktion vergleicht den Inhalt einer Adresse in einer gewünschten Bank mit dem Inhalt des Akkus. Es werden die bei einem Vergleich üblichen Flags gesetzt (z. B. Zero-Flag = 1 wenn beide gleich).
Die folgenden Speicherstellen müssen vor dem Aufruf gesetzt worden sein:

```
$02C8 : Low-Byte eines Vektors, der auf die gewünschte
        Adresse zeigt.
Akku  : Zeichen, das mit dem Inhalt der adressierten
        Speicherstelle verglichen werden soll.
X-Reg.: Nummer der gewünschten Bank
```

Durch die letzten sieben Routinen haben Sie die wichtigsten Routinen zum Umgang mit mehreren Banks erhalten. Tips & Tricks zu diesen Routinen erhalten Sie im nächsten Abschnitt.

Kernal-Adresse: $FF7D (65405)
Name: primm
Tatsächliche Sprungadresse: $FA17 (64023)
Funktion: Ausgeben eines Strings

Im alten Teil des Kernals existiert eine Routine, mit der man ein Zeichen ausgeben kann (BSOUT). Mit dieser Routine kann man nun einen String ausgeben und zwar auf ziemlich raffinierte Weise: Die auszugebenden Zeichen werden einfach hinter dem Aufruf der Routine plaziert, so daß sich folgendes Bild ergibt:

```
JSR $FF7D
(beliebiger Text, eine 0 als Endekennzeichen)
RTS
```

Doch wie kann das angehen, wie gelangt der Rechner denn überhaupt zu dem abschließendem RTS (oder was da sonst stehen mag)? Dazu werfen wir doch einmal einen Blick auf die Routine:

```
FA17  PHA
FA18  TXA
FA19  PHA
FA1A  TYA
FA1B  PHA
FA1C  LDY #$00
FA1E  TSX
FA1F  INC $0104,X
FA22  BNE $FA27
FA24  INC $0105,X
FA27  LDA $0104,X
FA2A  STA $CE
FA2C  LDA $0105,X
FA2F  STA $CF
FA31  LDA ($CE),Y
FA33  BEQ $FA3A
FA35  JSR $FFD2
FA38  BCC $FA1F
FA3A  PLA
FA3B  TAY
FA3C  PLA
FA3D  TAX
FA3E  PLA
FA3F  RTS
```

Zuerst einmal werden die Registerinhalte des Akkus, des X- und des Y-Registers auf den Stack gesichert. Dann wird das Y-Register, das später bei indirekt indizierter Adressierung (haben Sie auch schön die Namen der Adressierungsarten gelernt?) als Offset benutzt wird.
Nun beginnt der wichtige Teil der Routine: der Stackpointer, d. h. der Zeiger auf den nächsten freien Platz im Stack,

wird in das X-Register gespeichert, von wo aus er als Offset benutzt wird.
Das geschieht schon im nächsten Befehl. Um zu verstehen, was hier vor sich geht, muß man den Aufbau und die Lage des Stacks kennen. Beim 128er befindet er sich von $0139 - $01FF im Speicher und "wächst" von unten nach oben, d. h. der Stackpointer zeigt zuerst auf $01FF und wird bei einer größeren Anzahl von Elementen auf dem Stack immer kleiner.
Versuchen wir also einmal zu simulieren, wie sich der Stack bei Aufruf dieser Routine verhält. Nehmen wir an, der Stackpointer enthält den Wert $F0 und zeigt somit auf die Adresse $01F0. Der Stack sieht also folgendermaßen aus:

(Zeichnung 11.2.1.a)

Nun wird die Routine aufgerufen. Wir nehmen einmal an, daß der Aufruf in Adresse $2000 geschieht. Der Programm-Counter (16-Bit-Register, das auf die Adresse zeigt, aus der das nächste Byte geholt wird) wird auf dem Stack abgelegt, zuerst das High-, dann das Low-Byte.
In unserem Fall wird zuerst $20, dann $02 abgelegt, da aus $2002 das letzte Byte (High-Byte der Sprungadresse) gelesen wurde.
Der Stack sieht demnach folgendermaßen aus:

(Zeichnung 11.2.1.b)

Nun wird die Routine angesprungen, wo zuerst einmal die drei Register abgespeichert werden und sich dann folgendes Bild ergibt:

(Zeichnung 11.2.1.c)

In Adresse $FA1F wird nun ein Byte des Stapels angesprochen. Doch welches? Rechnen wir es doch einfach aus: Der Stapelzeiger (und damit auch das X-Register) enthält den Wert $EB, zu $0104 addiert ergibt das $01EF. Es wird also das Low-Byte der Rücksprungadresse um eins erhöht. Der BRANCH-Befehl kontrolliert, ob das Low-Byte Null geworden ist. Ist dies der Fall, so wird das Byte in Adresse $0105 +

$EB = $01F0 = High-Byte der Rücksprungadresse um eins erhöht.
Die nächsten vier Befehle setzen also den Vektor $CE / $CF auf das erste Byte des auszugebenden Strings.
Nun wird das Zeichen einfach eingelesen, geprüft ob es Null ist und ggf. ausgegeben. Durch den darauffolgenden BRANCH-Befehl werden alle Zeichen ausgegeben.
Ist die Schleife beendet, so werden die Registerinhalte wieder zurückgeholt und es ergibt sich folgendes Bild des Stacks:

(Zeichnung 11.2.1.d)

Schließlich gelangt der Prozessor zu dem RTS-Befehl und holt sich die Rücksprungadresse vom Stack. Diese wurde jedoch durch die vorhergehende Schleife soweit inkrementiert, daß sie auf das letzte Byte des Strings, also das Endekennzeichen Null, zeigt.
Nun erhöht der Prozessor noch den Programm-Zähler und führt weiter lustig Befehle aus.

Wie Sie sehen, haben sich die Programmierer ganz schön etwas einfallen gelassen. Was, Sie sehen die Vorteile dieser Routine nicht? Nun, man muß nicht wie sonst üblich die Register mit Zeigern auf den auszugebenden String laden. Dadurch wird das Programm kürzer und schneller. Und diese beiden Punkte, Schnelligkeit und Speicherbedarf, sind Probleme, mit denen sich nicht nur Systemprogrammierer rumschlagen...

Damit wären die Routinen des neuen Kernals auch schon ausgeschöpft und es geht weiter im nächsten Abschnitt.
Zuvor jedoch noch ein Wort zu den Namen der einzelnen Routinen: Sie wurden nicht von uns kreiert, sondern stammen aus den Systemunterlagen von Commodore.

11.2.2. Vektor-Lade-Tabelle

Im BASIC-Interpreter ist von $42CE - $4309 eine Tabelle zu finden, mit der man problemlos auf Speicherstellen zugreifen kann, die durch bestimmte Vektoren festgelegt sind.
Sie können diese Routinen also folgendermaßen benutzen: Sie schreiben die gewünschte Adresse in den Vektor und rufen dann die zu dem Vektor gehörende Routine auf.
Diese Tabelle hat leider jedoch auch ein paar Nachteile:

1. Es kann nicht jeder Vektor benutzt werden. Welche zur Verfügung stehen, erfahren Sie aus der Tabelle weiter unten.
2. Das Byte kann nicht von einer beliebigen Bank geholt werden.
3. Es wird nicht immer zu der ursprünglichen Bank zurückgesprungen.

Wollen Sie also von einer bestimmen Bank ein Byte holen und dann zu einer bestimmten Bank zurückkommen, so schauen Sie immer erst in der unten stehenden Tabelle nach, ob Ihre Kombination erhältlich ist. Sollte dies nicht der Fall sein, so müssen Sie wohl oder übel die Kernal-Routine indfet ($FF74) benutzen.
Nun die Tabelle:

Adresse	Vektor	Byte von	Nach dem Rücksprung
$42CE	$50/$51	Bank 1	Bank 14 (RAM-Bank 1)
$42D3	$3F/$40	Bank 1	Bank 14 (RAM-Bank 1)
$42D8	$52/$53	Bank 1	Bank 14 (RAM-Bank 1)
$42DD	$5C/$5D	Bank 0	Bank 14
$42E2	$5C/$5D	Bank 1	Bank 14 (RAM-Bank 1)
$42E7	$66/$67	Bank 1	Bank 14 (RAM-Bank 1)
$42EC	$61/$62	Bank 0	Bank 14
$42F1	$70/$71	Bank 0	Bank 14
$42F6	$70/$71	Bank 1	Bank 14 (RAM-Bank 1)
$42FB	$50/$51	Bank 1	Bank 14 (RAM-Bank 1)
$4300	$61/$62	Bank 1	Bank 14 (RAM-Bank 1)
$4305	$24/$25	Bank 0	Bank 14

"Bank 14 (RAM-Bank 1)" heißt, daß der Speicher nach dem

Umschalten die Konfiguration wie bei Bank 14 zeigt, nur wurde statt RAM-Bank 0 die RAM-Bank 1 eingeschaltet.
Bei den Routinen wird immer nur zwischen RAM und ROM geschaltet, niemals wird die RAM-Bank an sich verändert.

11.2.3. Kernal-Aufruf

Auch diese Tabelle ist im BASIC-Interpreter zu finden. Hier werden einige Kernal-Routinen aufgerufen, allerdings wird vorher Bank 15 eingeschaltet (mit einer Ausnahme: $928D; hier wird Bank 14 eingeschaltet).

Adresse :	Kernal-Routine
$9251	: Status holen ($FFB7)
$9257	: Fileparameter setzen ($FFBA)
$925D	: Filenamenparameter setzen ($FFBD)
$9263	: BASIN ($FFCF)
$9269	: BSOUT ($FFD2)
$926F	: CLRCH ($FFCC)
$9275	: CLOSE ($FFC3)
$927B	: CLALL ($FFE7)
$9281	: primm ($FF7D)
$9287	: setbnk ($FF68)
$928D	: Cursor setzen / holen ($FFF0)
$9293	: Stop-Taste abfragen ($FFE1)

11.3. Freier Speicher

Maschinenprogramme sind schön und gut, aber oft muß man sich beim Schreiben von derartigen Programmen fragen: Wo lege ich mein Programm hin? Je nach Sinn und Eigenschaften der Routine wird man einen anderen Speicherbereich suchen. So eignet sich der BASIC-RAM natürlich nur, wenn kein BASIC-Programm in den Speicher geschrieben werden soll (oder man setzt den BASIC-Anfang hoch bzw. das BASIC-Ende runter). Weiterhin ist es wichtig zu wissen, welche Adressen in der Zeropage benutzt werden können, da sich einerseits einige wichtige Adressierungsarten nur über Zeiger in der Zeropage anwenden lassen und man andererseits durch die Benutzung von Adressen in der Zeropage viel Zeit sparen kann.
Diese freien Bereiche sollen hier gezeigt werden.
Kommen wir erst einmal auf die Zeropage zu sprechen:

11.3.1. Verwendbare Adressen in der Zeropage

Völlig unbenutzt und damit immer frei sind in der Zeropage die Adressen $FA - $FE.
So gut wie frei sind noch die Adressen $4E / $4F. Soweit wir aus dem Betriebssystem und dem BASIC-Interpreter ersehen konnten, werden diese beiden Adressen nur von der boot-call-Routine (siehe Kapitel 10.1) benutzt.
Doch damit wären die freien Adressen in der Zeropage auch schon ausgeschöpft. Doch immerhin! Sechs Bytes sind doch gar nicht schlecht.
In den anderen Pages, die noch vom Betriebssystem belegt werden, existieren auch freie Bytes, es hätte jedoch keinen Sinn diese auszuzählen, da Sie statt diesen auch beliebige andere Adressen im Speicher benutzen können.
Kommen wir deshalb zum zweiten, dem freien Platz für Maschinenprogramme.

11.3.2. Verwendbarer Speicher für Maschinenprogramme

Beim C-64 waren zwei Speicherplätze gängig: der Kassettenpuffer für kurze Programme und der Bereich $C000 - $CFFF. Schon seltener kam es vor, daß ein Maschinensprache-Programm ans BASIC-Ende oder sogar an den BASIC-Anfang gesetzt wurde.
Beim C-128 gibt es ein paar mehr Möglichkeiten.
Zuerst einmal existiert natürlich auch wieder der Kassettenpuffer. Er geht hier von S0B00 - $0BFF.
Direkt hinter dem Kassettenpuffer liegen zwei Puffer, die bei RS232-Schnittstellen-Betrieb benutzt werden:

1. $0C00 - $0CFF = RS232 input buffer
2. $0D00 - $00FF = RS232 output buffer

Da die wenigsten C-128 Besitzer Daten über die RS232-Schnittstelle jagen werden, bietet sich dieser Bereich geradezu an. Man hat dann durchgehend von $0B00 - $0DFF die gigantische Anzahl von 768 Bytes für Maschinenprogramme zur Verfügung, zumal der Kassettenpuffer beim 128er wohl bedeutend weniger als beim 64er benutzt werden wird (CP/M mit Kassette?).

Doch das war bei weitem noch nicht alles. Unter normalen Umständen (das heißt, bei uns ist noch nie das Gegenteil eingetreten) steht Ihnen auch noch der Bereich von $1300 - S1BFF zur Verfügung. Das sind 9 Pages oder 2304 Byte allerfeinsten RAMs! Gedacht ist der Bereich für ROM-Cartridges.

Doch es gibt noch eine schöne Möglichkeit: Legen Sie Ihre Programme in die RAM-Bank 0. Diese ist normalerweise nur für Variablen gedacht. Als 64er-Besitzer wird man (fast) 64K für Variablen jedoch für etwas übertrieben halten und einige Bytes für eigene Maschinenprogramm abzwacken.
Wie das geht und was Sie dabei beachten müssen, erfahren Sie im nächsten Kapitel.

Wie Sie sehen, bietet der C-128 viele Möglichkeiten, seine Programme im Speicher zu verteilen. Interessant ist auch, daß Grafik erzeugt werden kann, ohne daß man auf kostbares RAM verzichten muß. Das ist möglich, da der VDC über einen eigenen RAM-Speicher von 16K verfügt. Wie das geht, erfahren Sie im Kapitel über den 80-Zeichen-Bildschirm.

12. WECHSELN DES BETRIEBSSYSTEMS

12. Wechseln des Betriebssystemes

****

Da der 128er nicht nur ein 128er, sondern auch ein 64er sein kann, muß es irgendetwas geben, das ihn in diesen Modus versetzt. In BASIC gibt es dafür den Befehl "GO 64". In der neuen Kernal-Sprungtabelle existiert eine Routine namens "c64mode". Diese Routine wollen wir uns einmal anschauen. Dazu zuerst das Listing:

```
E24B   LDA #$E3
E24D   STA $01
E24F   LDA #$2F
E251   STA $00
E253   LDX #$08
E255   LDA $E262,X
E258   STA $01,X
E25A   DEX
E25B   BNE $E255
E25D   STX $D030
E260   JMP $0002
----------------------
E263   LDA #$F7
E265   STA $D505
E268   JMP ($FFFC)
```

Die ersten vier Befehle schalten den Prozessor-Port um und sind für uns nicht so interessant.
Die nächsten fünf Befehle stellen eine Schleife dar. Hier werden die Adressen $E263 - $E26A nach $0002 - $0009 kopiert. Zum Zweck der ganzen Angelegenheit kommen wir gleich noch.
Der Befehl in Adresse $E25D stellt die Taktgeschwindigkeit auf 1 MHz um (wenn Sie das näher interessiert, schauen Sie bitte in das Kapitel 13.1). Dazu wird Null (das X-Register ist durch die vorhergehende Schleife auf 0 gesetzt worden) in das Register 48 des neuen VIC gespeichert.

In Adresse $E260 geht's nun los. Es werden der Reihe nach die durch die Schleife kopierten Befehle abgearbeitet. Das eigentliche Umschalten geschieht nun durch die ersten beiden Befehle. Die Speicherstelle $D505 gehört zur MMU und hat folgende Bedeutung:

Bit-Nummer	Funktion bei 0	Funktion bei 1
0	Z-80 ein	6502 ein
1 & 2	momentan nicht benutzt	
3	FSDIR controll Bit (für Floppy)	
4	C-64	C-128
5	C-64	C-128
6	C-128	C-64
7	40-Zeichen	80-Zeichen

Zur Erklärung:
Bit 3 & 4 sind nur lesbar; durch Schreiben wird nicht der Modus umgeändert. Diese beiden Bits stellen nur die Polarität zweier Leitungen am Expansions-Port dar:

Bit 4 : GAME
Bit 5 : EXROM

Diese beiden Leitungen werden bei einem Reset abgefragt. Ist eine der beiden auf logisch Null, so wird in den 64er-Modus gesprungen. Durch diese beiden Leitungen kann also erreicht werden, daß bei angeschlossenen 64er-Modulen sofort in den 64er-Modus gesprungen wird. Sie können sich natürlich auch eine entsprechende Taste einbauen, so daß Sie per Tastendruck in den 64er-Modus gelangen können.

Auch Bit 7 kann nur gelesen werden. Dieses Bit gibt den Zustand der Taste "40/80-Display" an. Ist es 1, so ist die Taste im gedrückten Zustand. Dieses Bit gibt also keine Auskunft über den Modus, in dem sich der Computer wirklich befindet. Wird z. B. durch "ESC + X" der Modus gewechselt, so verändert sich dieses Bit nicht.

Nun zu dem wichtigsten Bit: Bit 6. Ist es auf 1, so geht der

C-128 in den 64er-Modus. Nun können wir auch die beiden Befehle in der c64mode-Routine verstehen. Der hexadezimale Wert $F7 entspricht dem binären Wert 11110111. Bit 6 wird also gesetzt.

Mit dem letzten Befehl wird schließlich ein Reset durchgeführt.

Ist es Ihnen aufgefallen? Nach dem Umschalten, d. h. bevor der Reset ausgeführt wird, werden alle ROM's des C-128 aus- und die des 64er eingeschaltet. Würden nun die letzten drei Befehle nicht im RAM der Bank 0 (die von beiden Rechnern benutzt wird), sondern im ROM oder im RAM der Bank 1 liegen, so würde sich der Rechner aufhängen. Sie glauben uns nicht? Probieren Sie es aus: Springen Sie die Routine bei $E263 an. Die Befehle liegen jedoch im RAM und so macht der Prozessor, nun ein 6510, bei Adresse $0007 weiter (der Programmzähler wurde beim Umschalten natürlich beibehalten) und führt einen ordnungsgemäßen (64er-)Reset aus.

Tüfteln wir nun ein bißchen rum.
Zuerst einmal wollen wir beweisen, daß die RAM-Bank 0 wirklich von beiden Rechnern benutzt wird. Geben Sie also folgendes ein:

```
POKE 8192,170: POKE 8193,85

GO 64 (und dann natürlich 'Y' für YES)

PRINT PEEK(8192);PEEK(8193)
```

Sie erhalten die von Ihnen eingegebenen Werte wieder auf den Bildschirm zurück.

Doch halt! Wenn der Speicher nicht gelöscht wird, so müßten doch auch ganze Programme dem 64er übergeben werden können! Geht auch. Geben Sie zuerst einmal ein kleines Programm ein. Wechseln Sie nun wieder mit "GO 64" den Modus. Jetzt müssen Sie folgende POKEs eingeben:

```
POKE 43,1: POKE 44,28
```

Und schon kommt Ihr Programm wieder zum Vorschein. Sie können es sogar wieder ablaufen lassen. Etwas müssen Sie dabei allerdings beachten: In dem Programm dürfen keine Befehle verwendet werden, die der C-64 nicht beherrscht.

Es ist sogar möglich, ein Maschinenprogramm, daß im C-128 gestartet wurde, ohne Unterbrechung im 64er-Modus weiterzuführen. Dabei muß allerdings der größte Teil der Reset-Routine vorher angesprungen worden sein, da sonst ziemlich viel falsch gehen würde (so wäre der IRQ-Vektor nicht definiert, der Bildschirm wäre nicht auf seine Normal-Werte gesetzt etc., etc.).
Ein Beispiel:

```
2000 A9 F7    LDA #$F7     ; Umschalten
2002 80 05 05 STA $D505    ; in 64er-Modus
2005 A2 FF    LDX #$FF     ; Reset-Routine
2007 78       SEI          ; bis auf Sprung
2008 9A       TXS          ; zum BASIC-Kaltstart
2009 D8       CLD          ; und Test auf ROM in $8000
200A 8E 16 D0 STX $0016
2000 20 A3 FD JSR $FDA3
2010 20 50 FD JSR $FD50
2013 20 15 FD JSR $FD15
2016 20 5B FF JSR $FF5B
2019 58       CLI
201A EE 20 D0 INC $0020    ; Hintergrundfarbe erhöhen
201D 4C 1A 20 JMP $        ; und noch einmal
```

Rufen Sie nun die Routine mit SYS DEC("2000") auf. Nach einem kurzen Moment erhalten Sie einen bunten Rahmen - im 64er-Modus, ohne das Programm zu unterbrechen.

Leider ist das mit BASIC-Programmen nicht so einfach. Probieren Sie es doch einmal!

13. DER 64ER-MODUS AUF DEM C-128

13. Der C64-Mode auf dem Commodore 128

****

Ein wirklich nicht zu unterschätzender Vorteil des Commodore 128 ist seine Kompatibilität zum Commodore 64. Wenn Sie also den Rechner in den C64-Mode umschalten, kommen Sie in den Genuß sämtlicher auf dem Markt befindlicher C64-Software. Wir brauchen Ihnen sicher nicht zu sagen, wie groß dieses Angebot ist!

Die Commodore-Leute mußten also dafür sorgen, daß alle C64-Programme in der Betriebsart C64 auch wirklich laufen. So stellt der C64-Mode im Grunde nichts weiter dar als eine regelrechte Umschaltung der gesamten Speicherkonfiguration des C128 zu einem C64.
Im C64-Mode steht Ihnen also auch nur das "normale" V,2 BASIC zur Verfügung. Auch ein Bank-Switching oder der Z80-Prozessor werden damit ausgeschaltet.

Aber im C-64-Modus ist der C-128 kein "normaler" C-64. Einige Features des C-128 wurden dem C-64 (glücklicherweise) mitgegeben:

13.1 High-Speed für den C-64

Wie bereits im Kapitel "Der 40-Zeichen-Bildschirm" angeschnitten, verfügt der C128 zur Darstellung des 40-Zeichen Bildschirmes über den Video Controller 8564. Im Commodore 64 war hingegen ein VIC 6564. Wie ist das zu erklären?

Der neue VIC ist "aufwärtskompatibel" zum vom C64 her gewohnten VIC 6564. Das bedeutet, er erfüllt im Prinzip die gleichen Aufgaben wie der alte VIC, ist nur in bestimmten Bereichen erweitert worden. Konkret: Es sind zwei neue Register hinzugekommen. Diese neuen Register können auch aus dem C64-Mode heraus manipuliert werden!
Das letztere der beiden, Register 48, spielt dabei eine besondere Rolle:

48 Rechner-Takt

Wie auch im C64 ist hier dieser Chip für die Taktversorgung des ganzen Rechners zuständig. Mit diesem Register nun kann die Taktgeschwindigkeit von 1 MHz auf 2 MHz verdoppelt werden; das funktioniert auch im C64-Modus!

Alle Ihre "alten" C64-Programme können so mit der doppelten Geschwindigkeit ablaufen. Gerade abendfüllende Rechenroutinen dürften so etwas aufgefrischt werden! Der Geschwindigkeitszuwachs beschleunigt aber auch zeitintensive Systemroutinen wie die Garbage Collection (ein Greuel jedes Datei-Programmierers!) um das Doppelte.
Allerdings hat die Sache auch einen Haken: Der VIC benutzt die Taktlücken normalerweise im System dazu, sich schnell ein Zeichen aus dem Video-RAM zu schnappen, um das Bild wieder aufzufrischen. Wird der Takt jetzt verdoppelt, ist diese Lücke nur noch halb so lang. Das ist wiederum viel zu kurz für den VIC, der nun das Bild nicht mehr auffrischen kann, was zur Folge hat, daß sich bei doppelter Geschwindigkeit auf dem Bildschirm die merkwürdigsten Dinge tun - nur nicht das, was man eigentlich erwartet.
Sie sollten deshalb nur die Programmteile mit doppeltem Takt ablaufen lassen, bei denen der Bildschirm für den Anwender keine Rolle spielt. Hierzu zählen beispielsweise rechen- und zeitintensive Programmteile. Um Ergebnisse über den Bildschirm auszugeben, muß wieder der normale Takt eingestellt werden. Dies sieht im Prinzip so aus:

```
POKE 53296,1  (2 MHz, schnell)
POKE 53296,0  (1 MHz, normal)
```

Um Ihnen mal zu zeigen, was das bringt, folgendes Programm:

```
10 TI$="000000"
20 :
30 FOR X=1 TO 10000
40 NEXT X
```

```
50 :
60 PRINT TI$
```

Dieses Programm benötigt so, wie es ist, volle 14 Sekunden für die Schleife. Schneller gehts mit den zusätzlichen Zeilen:

```
20 POKE 53296,1:REM SCHNELL
50 POKE 53296,0:REM NORMAL
```

Die gleiche Schleife benötigt jetzt nur noch 7 Sekunden!

13.2. Der Zugriff auf den 80-Zeichen-Controller

Ja, Sie haben richtig gelesen, auch im 64er-Modus kann auf den 80-Zeichen-Controller zugegriffen werden! Und das ohne irgendwelche Tricks anzuwenden: Der Zugriff geschieht nach den gleichen Regeln wie im 128er-Modus (siehe Kapitel 1).

Dieses Bonbon von Commodore muß natürlich weidlich ausgenutzt werden: endlich volle 80 Zeichen pro Zeile. Natürlich ist das Betriebssystem des C-64 nicht auf diesen Controller eingestellt. Dies abzuändern dürfte jedoch kein Problem sein, da ja nur die entsprechenden Routinen des C-128-Betriebssystemes übernommen werden müssen.

Beispiele würden leider den Rahmen dieses Buches sprengen. Nur noch einige Anregungen: Wie wäre es mit einer Grafikerweiterung, mit der man im 64er-Modus den 80-Zeichen-Bildschirm ansprechen kann? Oder eine Betriebssystem-Erweiterung (z. B. im IRQ gelegen), so daß auch auf den 80-Zeichen-Bildschirm geschrieben werden kann?

13.3. Zehnertastatur am 64er

Dies ist eher eine Anregung als ein Tip: Durch einen einfachen POKE vermag man die 10er-Tastatur im 64er-Mode zu aktivieren! Leider werden nur irgendwelche, aber nicht die

richtigen Zeichen ausgegeben. Da wir nicht genügend Zeit besaßen, konnten wir uns nicht näher mit diesem Phänomen auseinandersetzen. Versuchen Sie doch einmal, die 10er-Tastatur vollständig in Betrieb zu nehmen!

Nun der POKE:

POKE 53295,248

Zur Erklärung: Die Speicherstelle 53295 entspricht übrigens dem Register 47 des neuen VIC - eines der beiden neuen Register. In diese Speicherstelle wird einfach der Wert gespeichert, den sie im 128er-Modus besitzt.

14. ANHANG

14. Anhang

****

14.1. Token-Tabelle

BASIC-Befehle werden nicht als Zeichenfolge im BASIC-RAM abgelegt, sondern in sogenannte TOKEN umgewandelt. Ein spezielles Unterprogramm im BASIC-Interpreter schaut bei der Eingabe einer BASIC-Zeile nach, ob in dieser Zeile ein bekannter Befehl vorliegt. Findet er einen, so schaut er in einer Tabelle nach und setzt statt der Zeichenfolge einen bestimmten Wert. Statt den Bytes 82, 85, 78 für "RUN" wird das Byte 138 ($8A) eingesetzt.
Probieren Sie es aus: Tippen Sie NEW und dann die Zeile "10 RUN" ein. Gehen Sie nun in den Monitor und geben Sie "M 1C00" ein. Nun erhalten Sie einen Auszug des BASIC-RAM-Anfanges. Doch wo ist das RUN geblieben? Wenn Sie genau hinschauen, entdecken Sie den Wert $8A, den Token für RUN.
Doch warum das ganze? Zum einen wird dadurch viel BASIC-Speicher gespart. Nehmen wir z. B. den Befehl "DIRECTORY". Zeichenweise abgespeichert würde er 9 Bytes verbrauchen. Durch die Token-Umwandlung ist es nur noch ein Byte. Andererseits wird so das BASIC-Programm auch schneller. Es müssen nicht mehr 9, sondern nur noch ein Byte verglichen werden. Und die Umwandlung macht sich zeitlich nicht bemerkbar, da sich der Computer während der Eingabe von BASIC-Zeilen sowieso die meiste Zeit in der Eingabe-Warteschleife befindet.
Die Token fangen übrigens erst bei $80 (128) an. Das hat den Vorteil, daß ein Zeichen schnell darauf getestet werden kann, ob es einen BASIC-Befehl darstellt. Es muß nur Bit 7 mit dem BIT-Befehl getestet werden. Es existiert jedoch auch ein Nachteil: Auf diese Art können nur 128 BASIC-Befehle dargestellt werden.
Wer die Token-Tabelle vom 64er kennt, weiß, daß noch ziemlich viele Token frei waren. Beim 128er sind diese restlos belegt. Das war jedoch immer noch nicht genug für den großen Befehlsschatz dieser BASIC-Version. Was nun? Man

ist auf einen einfachen Trick verfallen: Man hat einfach mehrere Token doppelt belegt. Um die Befehle trotzdem noch voneinander unterscheiden zu können, wurde hinter diese Token noch ein Byte gehängt. So hat der Befehl "BANK" z. B. den Token $FE + $02. Durch diese Methode können natürlich ganz schön viele Befehle implementiert werden.

Nun die Token-Tabelle:

Wert (hex.)	Wert (dez.)	BASIC-Befehl
$80	128	END
$81	129	FOR
$82	130	NEXT
$83	131	DATA
$84	132	INPUT#
$85	133	INPUT
$86	134	DIM
$87	135	READ
$88	136	LET
$89	137	GOTO
$8A	138	RUN
$8B	139	IF
$8C	140	RESTORE
$8D	141	GOSUB
$8E	142	RETURN
$8F	143	REM
$90	144	STOP
$91	145	ON
$92	146	WAIT
$93	147	LOAD
$94	148	SAVE
$95	149	VERIFY
$96	150	DEF
$97	151	POKE
$98	152	PRINT#
$99	153	PRINT
$9A	154	CONT

$9B	:	155	:	LIST
$9C	:	156	:	CLR
$90	:	157	:	CMD
$9E	:	158	:	SYS
$9F	:	159	:	OPEN

$A0	:	160	:	CLOSE
$A1	:	161	:	GET
$A2	:	162	:	NEW
$A3	:	163	:	TAB(
$A4	:	164	:	TO
$A5	:	165	:	FN
SA6	:	166	:	SPC(
$A7	:	167	:	THEN
$A8	:	168	:	NOT
$A9	:	169	:	STEP
$AA	:	170	:	+
SAB	:	171	:	-
$AC	:	172	:	*
SAD	:	173	:	/
$AE	:	174	:	^
$AF	:	175	:	AND

$80	:	176	:	OR
$B1	:	177	:	<
$B2	:	178	:	=
$B3	:	179	:	>
$B4	:	180	:	SGN
SB5	:	181	:	INT
$B6	:	182	:	ABS
$B7	:	183	:	USR
$B8	:	184	:	FRE
$B9	:	185	:	POS
$BA	:	186	:	SQR
$BB	:	187	:	RND
$BC	:	188	:	LOG
$BD	:	189	:	EXP
$BE	:	190	:	COS
$BF	:	191	:	SIN

$C0	:	192	:	TAN
$C1	:	193	:	ATN
$C2	:	194	:	PEEK
$C3	:	195	:	LEN
$C4	:	196	:	STR$
$C5	:	197	:	VAL
$C6	:	198	:	ASC
$C7	:	199	:	CHR$
$C8	:	200	:	LEFT$
$C9	:	201	:	RIGHT$
$CA	:	202	:	MID$
$CB	:	203	:	GO
$CC	:	204	:	RGR
$CD	:	205	:	RCLR
$CE	:	206	:	siehe unten
$CF	:	207	:	JOY

$00	:	208	:	RDOT
$01	:	209	:	DEC
$D2	:	210	:	HEX$
$D3	:	211	:	ERR$
$D4	:	212	:	INSTR
$D5	:	213	:	ELSE
$D6	:	214	:	RESUME
$D7	:	215	:	TRAP
$D8	:	216	:	TRON
$09	:	217	:	TROFF
$DA	:	218	:	SOUND
$DB	:	219	:	VOL
$DC	:	220	:	AUTO
$DD	:	221	:	PUDEF
$OE	:	222	:	GRAPHIC
$DF	:	223	:	PAINT

$EO	:	224	:	CHAR
$E1	:	225	:	BOX
$E2	:	226	:	CIRCLE
$E3	:	227	:	GSHAPE
$E4	:	228	:	SSHAPE
$E5	:	229	:	DRAW

$E6	:	230	:	LOCATE
$E7	:	231	:	COLOR
$E8	:	232	:	SCNCLR
$E9	:	233	:	SCALE
$EA	:	234	:	HELP
$EB	:	235	:	DO
$EC	:	236	:	LOOP
$ED	:	237	:	EXIT
$EE	:	238	:	DIRECTORY
$EF	:	239	:	DSAVE
$F0	:	240	:	DLOAD
$F1	:	241	:	HEADER
$F2	:	242	:	SCRATCH
$F3	:	243	:	COLLECT
$F4	:	244	:	COPY
$F5	:	245	:	RENAME
$F6	:	246	:	BACKUP
$F7	:	247	:	DELETE
$F8	:	248	:	RENUMBER
$F9	:	249	:	KEY
$FA	:	250	:	MONITOR
$FB	:	251	:	USING
$FC	:	252	:	WHILE
$FD	:	253	:	UNTIL
$FE	:	254	:	mehrfach
$FF	:	255	:	pi

Nun kommen die Tabellen für die Token, die mehrfach belegt worden sind. Es sind derer zwei: $CE und $FE.
Die Werte beziehen sich diesmal auf das Byte hinter dem Token.

Tabelle für Befehle mit $CE als erstem Token:

Wert (hex.)	:	Wert (dez.)	:	BASIC-Befehl
$00	:	0	:	RWINDOW

$01	:	1	: nicht belegt
$02	:	2	: POT
$03	:	3	: BUMP
$04	:	4	: PEN
$05	:	5	: RSPPOS
$06	:	6	: RSPRITE
$07	:	7	: RSPCOLOR
$08	:	8	: XOR

Tabelle für Befehle mit $FE als erstem Token:

Wert (hex.)	:	Wert (dez.)	: BASIC-Befehl
$00	:	0	: nicht belegt
$01	:	1	: nicht belegt
$02	:	2	: BANK
$03	:	3	: FILTER
$04	:	4	: PLAY
$05	:	5	: TEMPO
$06	:	6	: MOVSPR
$07	:	7	: SPRITE
$08	:	8	: SPRCOLOR
$09	:	9	: RREG
$0A	:	10	: ENVELOPE
$0B	:	11	: SLEEP
$0C	:	12	: CATALOG
$0D	:	13	: DOPEN
$0E	:	14	: APPEND
$0F	:	15	: DCLOSE
$10	:	16	: BSAVE
$11	:	17	: BLOAD
$12	:	18	: RECORD
$13	:	19	: CONCAT
$14	:	20	: DVERIFY
$15	:	21	: DCLEAR
$16	:	22	: SPRSAV
$17	:	23	: COLLISION
$18	:	24	: BEGIN

$19	:	25	:	BEND
$1A	:	26	:	WINDOW
$1B	:	27	:	BOOT
$1C	:	28	:	WIDTH
$1D	:	29	:	SPRDEF

15. DIE BASIC-LISTINGS

(PRG 1.3.A) Auslesen des Zeichengenerators

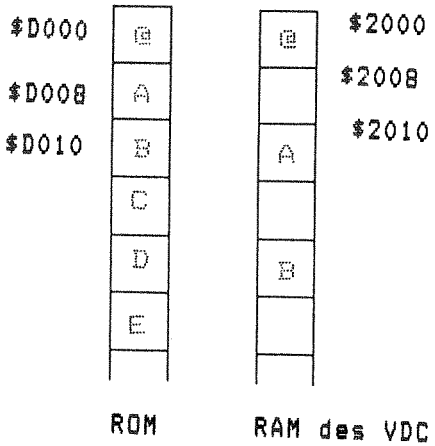
```
90 PRINT"] AUSLESEN DES ZEICHENGENERATORS"
91 PRINT:PRINT
100 REM CHARGEN AUSLESEN
120 :
130 FOR X=0 TO 255:REM 256 ZEICHEN AUSLESEN
140 FOR Z=0 TO 7:REM 8 BYTES
150 AD=53248+X*8+Z
180 BANK 14:C=PEEK(AD):BANK 15
210 :
220 FOR Y=7 TO 0 STEP -1:REM 8 PIXELZEILEN/ZEICHEN
230 IF C>=2↑Y THEN C=C-2↑Y:PRINT"*";:ELSE PRINT ".";
240 NEXT Y
250 :
260 PRINT
270 NEXT Z
280 PRINT
290 NEXT X
300 END
```

(PRG 1.10.A) Modifizierter POKE-Befehl

```
10 FOR X= 6144 TO 6182
20 READ A:CS=CS+A:POKE X,A
30 NEXT X
40 IF CS<> 3411 THEN PRINT CHR$(7);:LIST
50 DATA  72, 138, 72, 152, 72, 169, 2, 141, 40, 10, 162, 18
60 DATA  104, 32, 27, 24, 232, 104, 32, 27, 24, 162, 31, 104
70 DATA  76, 27, 24, 142, 0, 214, 44, 0, 214, 16, 251, 141
80 DATA  1, 214, 96
```

(PRG 1.12.A) Auslesen des Zeichengenerators

```
10 REM AUSLESEN DES ZEICHENGENERATORS
20 BA=8192
30 A=DEC("D600"): B=A+1
40 FOR X=0 TO 7
50 AD=BA+X+8*W: IF AD>16383 THEN END
60 HI=INT(AD/256): LO=AD-(256*HI)
70 POKE A,18: POKE B,HI
80 POKE A,19: POKE B,LO
90 POKE A,31: CH=PEEK(B)
100 FOR Y=7 TO 0 STEP -1
110 IF CH>=2↑Y THEN CH=CH-2↑Y: PRINT"*";: ELSE PRINT".";
120 NEXT Y
130 PRINT
140 NEXT X
150 W=W+1
160 GOTO 40
```



(Zeichnung 1.12.a)

(PRG 1.13.A) Big Script

```
10 Z$="*":GOSUB 60000:PRINTZ$:END
60000 Z=ASC(Z$):Z$="":IF Z AND 128 THEN X=Z AND 127 OR 64:GOTO 60040
60010 IF NOT Z AND 64 THEN X=Z:GOTO 60040
60020 IF Z AND 32 THEN X=Z AND 95:GOTO 60040
60030 X=Z AND 63
60040 L$="":U$=""
60050 FOR W=0 TO 7
60060 L$=L$+CHR$(157)
60070 U$=U$+CHR$(145)
60080 NEXT W
60090 D$=CHR$(17)
60100 REM CHARGEN AUSLESEN
60110 A=DEC("D600"):B=A+1
60120 :
60130 FOR Z=0 TO 7:REM 8 BYTES
60140 AD=8192+X*16+Z
60150 HI=INT(AD/256):LO=AD-(256*HI)
60160 :
60170 POKE A,18:POKE B,HI
60180 POKE A,19:POKE B,LO
60190 POKE A,31:C=PEEK(B)
60200 :
60210 FOR Y=7 TO 0 STEP -1:REM 8 PIXELZEILEN/ZEICHEN
60220 IF C>=2↑Y THEN C=C-2↑Y:ZE$=ZE$+"*":ELSE ZE$=ZE$+" "
60230 NEXT Y
60240 Z$=Z$+ZE$+L$+D$:ZE$=""
60250 NEXT Z
60260 Z$=LEFT$(Z$,LEN(Z$)-9)+U$
60270 RETURN
```

(PRG 1.14.A) Transparent-Drucker

```
10 S$="*":L$=" "
20 REM TRANSPARENT-DRUCKER
30 :
40 PRINT "■■ TRANSPARENT-DRUCKER ■■"
50 PRINT "■■■MIT DIESEM PROGRAMM WIRD DER GANG"
60 PRINT "■ZUR DEMO EINFACH!"
70 PRINT "■■■BITTE SCHALTEN SIE DEN DRUCKER"
80 PRINT "■■EIN UND DRUECKEN SIE 'RETURN' !"
90 GETKEY A$
100 OPEN 4,4
110 PRINT"■■ TRANSPARENT-DRUCKER ■■"
120 PRINT"■■FAKTOR-BESTIMMUNG:■■"
130 INPUT "■■HOEHE (1-10) ";HO
140 INPUT "BREITE  (1-...)";BR
150 PRINT"GEBEN SIE NUN DEN DRUCKTEXT AN!■■"
160 POKE 208,0:GETKEY A$:PRINT A$;
170 :
180 AC=ASC(A$)
190 IF AC AND 128 THEN CO=AC AND 127 OR 64:GOTO 230
200 IF NOT AC AND 64 THEN CO=AC:GOTO 230
210 IF AC AND 32 THEN CO=AC AND 95:GOTO 230
220 CO=AC AND 63
230 REM ** ZEICHENCODES AUSLESEN **
240 BANK 14
250 FOR A=0 TO 7
260 CH(A)=PEEK(53248+(8*CO)+A)
270 NEXT A
280 BANK 15
290 :
300 REM ** ZEICHEN DREHEN **
310 FOR A=0 TO 7
320 K(A)=0
330 NEXT A
340 FOR A=0 TO 7
350 FOR B=7 TO 0 STEP -1
360 W=2↑B
370 IF CH(A)>=W THEN CH(A)=CH(A)-W:K(7-B)=K(7-B)+2↑A
380 NEXT B,A
390 :
400 REM ** ZEICHEN AUSGEBEN **
410 FOR I=0 TO 7
420 Q$=""
430 FOR J=7 TO 0 STEP -1
440 WI=K(I)AND2↑J
450 IF WI THEN FOR U=1 TO HO:Q$=Q$+S$:NEXT U:GOTO 470
460 FOR U=1 TO HO:Q$=Q$+L$:NEXT U
470 NEXT J
480 REM ** UNNOETIGE LEERZEICHEN KUERZEN **
490 LX=LEN(Q$)-1
500 IF RIGHT$(Q$,1)=" " THEN Q$=LEFT$(Q$,LX):GOTO 490
510 FOR U=1 TO BR
520 PRINT#4,Q$
530 NEXT U
540 NEXT I
550 GOTO 160
```

(PRG 1.16.1) Zeichen-Editor

```
1 ZN=5100
2 MA=7
3 DIM D(MA),W$(MA)
10 D$="........"
20 REM ZEICHEN-EDITOR (80-ZEICHEN-CHARSET)
30 FOR Y=0 TO 7:D(Y)=0:NEXT Y
31 PRINT"■■■ VERAENDERN DES HAUSEIGENEN ZEICHENSATZES ■"
32 MF=1
40 PRINT
50 PRINT "▙WELCHES ZEICHEN WOLLEN SIE VERAENDERN ? -->  ";:GETKEY C$
55 PRINT C$
60 AC=ASC(C$)
70 IF AC AND 128 THEN CO=AC AND 127:GOTO 110
80 IF NOT AC AND 64 THEN CO=AC:GOTO 110
90 IF AC AND 32 THEN CO=AC AND 95:GOTO 110
100 CO=AC AND 63
110 AD=53248+CO*8
120 C$=""
130 :
140 PRINT "■■■ ORIGINAL ■        ■■■   USER   ■"
150 BANK 14
160 FOR X=0 TO MA
170 CZ=PEEK(AD+X):IF X>7 THEN CZ=0
180 FOR Y=7 TO 0 STEP -1
190 IF CZ>=2↑Y THEN C$=C$+"■*■":CZ=CZ-2↑Y:ELSE C$=C$+"."
200 NEXT Y
210 PRINT "■";C$;X;TAB(16);"■";:PRINT USING "##";X;
211 PRINT TAB(19);D$
220 C$=""
230 NEXT X
231 PRINT "■■■          ■        ■■■          ■"
232 BANK 15
240 :
260 REM EDITOR-ROUTINE
270 PRINT"■";
271 OPEN 1,0
280 FOR Y=0 TO MA
290 CHAR ,16,5+Y
300 PRINT " ->";
301 WINDOW 19,5+Y,27,5+Y,MF
302 POKE 244,1:INPUT#1,W$(Y)
310 W$(Y)=LEFT$(W$(Y),8)
311 WINDOW 0,0,79,24
320 FOR X=1 TO 8
330 IF MID$(W$(Y),X,1)="*" THEN D(Y)=D(Y)+2↑(8-X)
340 NEXT X
350 NEXT Y
351 CLOSE 1
360 CHAR,0,22:PRINT"KORREKTUR (J/N) ?";:GETKEY A$
370 IF A$="J" THEN MF=0:FOR Y=0 TO 7:D(Y)=0:NEXT Y:GOTO 260
380 PRINT:PRINT"UEBERNEHMEN (J/N) ?";:GETKEY A$
390 IF A$="N" THEN 30
400 GOTO 1000:REM UEBERNEHMEN
```

```
1000 PRINT"[illegible]";
1010 ZN=ZN+10
1020 PRINT ZN;"DATA ";CO;",";
1030 FOR X=0 TO MA:PRINT D(X);"[illegible],";:NEXT X
1031 PRINT"[illegible] "
1032 PRINT ZN+10;"DATA -1"
1039 PRINT:PRINT"GOTO 30"
1040 PRINT"[illegible]";
1050 FOR Y=842 TO 845:POKE Y,13:NEXT Y
1060 POKE 208,4
1070 END
5000 REM BASIC-LOADER ZEICHENDEFINITION
5005 FOR X=6144 TO 6182
5010 READ A
5015 CS=CS+A
5020 POKE X,A
5025 NEXT X
5030 IF CS<>3411 THEN PRINT CHR$(7):LIST 5035-5050:END
5035 DATA 72,138,72,152,72,169,2,141,40,10,162,18
5040 DATA 104,32,27,24,232,104,32,27,24,162,31,104
5045 DATA 76,27,24,142,0,214,44,0,214,16,251,141,1,214,96
5050 READ A:IF A=-1 THEN END
5055 FOR Y=0 TO 7
5060 READ W
5065 AD=8192+16*A+Y
5070 HI=INT(AD/256):LO=AD-(256*HI)
5075 SYS DEC("1800"),W,LO,HI
5080 NEXT Y
5085 GOTO 5050
```

(PRG 1.17.A) Zwei weitere Bildschirme

```
10 FOR X= 6912 TO 7008
20 READ A:CS=CS+A:POKE X,A
30 NEXT X
40 IF CS<> 9318 THEN PRINT CHR$(7);:LIST
50 DATA  120, 169, 27, 160, 24, 141, 21, 3, 140, 20, 3, 88
60 DATA  169, 0, 141, 0, 16, 141, 2, 16, 141, 4, 16, 96
70 DATA  72, 138, 72, 165, 213, 201, 88, 240, 58, 162, 12, 201
80 DATA  4, 240, 13, 201, 5, 240, 14, 201, 6, 208, 44, 169
90 DATA  0, 76, 62, 27, 169, 16, 76, 62, 27, 169, 24, 76
100 DATA  62, 27, 142, 0, 214, 44, 0, 214, 16, 251, 141, 1
110 DATA  214, 141, 46, 10, 162, 13, 169, 0, 142, 0, 214, 44
120 DATA  0, 214, 16, 251, 141, 1, 214, 104, 170, 104, 76, 101
130 DATA  250
```

(PRG 1.18.A) Explosionssimulation

```
10 REM EXPLOSIONSSIMULATION
15 A=DEC("D600"):B=A+1
20 FOR X=0 TO 50
30 Y=INT(RND(1)*2)+101
40 Z=INT(RND(1)*2)+31
50 POKE A,2:POKE B,Y
60 POKE A,7:POKE B,Z
70 NEXT X
80 POKE A,2:POKE B,102
90 POKE A,7:POKE B,32
```

(PRG 1.20.A) Änderung des Bildschirmrahmens

```
10 A=DEC("D600"):B=A+1
20 INPUT X,Y
30 POKE A,34:POKE B,X
40 POKE A,35:POKE B,X-Y
60 GOTO 20
```

(PRG 1.20.B) Verschieben des Rahmens

```
10 A=DEC("D600"):B=A+1
20 FOR X=0 TO 40
30 POKE A,34:POKE B,46-X
40 POKE A,35:POKE B,46+X
50 FOR T=1 TO 10:NEXT T
60 NEXT X
```

(PRG 1.21.A) Verkleinerung der Punktmatrix

```
10 A=DEC("D600"):B=A+1
20 FOR X=0 TO 8
30 POKE A,22:POKE B,PEEK(B)ANDNOT7ORX
40 FOR T=1 TO 100:NEXT T
50 NEXT X
60 FOR X=0 TO 8
70 POKE A,23:POKE B,X
71 FOR T=1 TO 100:NEXT T
80 NEXT X
90 END
```

(PRG 1.21.B) Demonstration der Verkleinerung der Punktmatrix

```
10 A=DEC("D600"):B=A+1
20 FOR X=0 TO 8
30 POKE A,22:POKE B,PEEK(B)ANDNOT7ORX
40 POKE A,23:POKE B,X
50 FOR T=1 TO 200:NEXT T
60 NEXT X
70 END
```

(PRG 1.23.A) 16*8-Punktematrix

```
10 REM POKE-ROUT
20 FOR X=6144 TO 6182
30 READ A:CS=CS+A:POKE X,A
40 NEXT X
50 IF CS<>3411 THEN PRINT "DATA-FEHLER":END
60 DATA 72,138,72,152,72,169,2,141,40,10,162,18
70 DATA 104,32,27,24,232,104,32,27,24,162,31,104
80 DATA 76,27,24,142,0,214,44,0,214,16,251,141,1,214,96
90 REM VERDOPPELN !
100 FOR W=0 TO 255:REM 256 ZEICHEN
110 FOR K=0 TO 7:REM 8 REIHEN LESEN
120 AD=53248+W*8+K
130 BANK 14:K(K)=PEEK(AD):BANK 15
140 NEXT K
150 FOR K=0 TO 15:REM 15 ZEILEN SETZEN
160 AD=8192+W*16+K
170 HI=INT(AD/256):LO=AD-(256*HI)
180 SYS DEC("1800"),K(INT(K/2)),LO,HI
190 NEXT K
200 NEXT W
```

(PRG 1.23.B) BASIC-Loader zur 16*8-Punktematrix

```
5000 FOR X= 2816 TO 2924
5010 READ A:CS=CS+A:POKE X,A
5020 NEXT X
5030 IF CS<> 13689 THEN PRINT CHR$(7);:LIST
5040 DATA  162, 3, 189, 65, 11, 149, 250, 202, 16, 248, 162, 1
5050 DATA  142, 0, 255, 160, 0, 177, 250, 72, 162, 0, 142, 0
5060 DATA  255, 166, 252, 164, 253, 32, 70, 11, 230, 252, 208, 2
5070 DATA  230, 253, 104, 166, 252, 164, 253, 32, 70, 11, 230, 252
5080 DATA  208, 2, 230, 253, 230, 250, 208, 2, 230, 251, 164, 251
5090 DATA  192, 224, 144, 202, 96, 0, 208, 0, 32, 0, 72, 138
5100 DATA  72, 152, 72, 169, 2, 141, 40, 10, 162, 18, 104, 32
5110 DATA  97, 11, 232, 104, 32, 97, 11, 162, 31, 104, 76, 97
5120 DATA  11, 142, 0, 214, 44, 0, 214, 16, 251, 141, 1, 214
5130 DATA  96
```

(PRG 1.26.A) Zusätzlicher Zeichengenerator

```
5000 FOR X= 2816 TO 2985
5010 READ A:CS=CS+A:POKE X,A
5020 NEXT X
5030 IF CS<> 17057 THEN PRINT CHR$(7);:LIST
5040 DATA  169, 0, 133, 76, 169, 32, 133, 77, 169, 8, 133, 78
5050 DATA  169, 32, 133, 79, 160, 7, 152, 72, 165, 76, 166, 77
5060 DATA  32, 140, 11, 72, 165, 78, 166, 79, 32, 140, 11, 166
5070 DATA  76, 164, 77, 32, 101, 11, 104, 166, 78, 164, 79, 32
5080 DATA  101, 11, 230, 76, 208, 2, 230, 77, 230, 78, 208, 2
5090 DATA  230, 79, 104, 168, 136, 16, 207, 165, 79, 201, 64, 176
5100 DATA  27, 169, 8, 101, 76, 133, 76, 169, 0, 101, 77, 133
5110 DATA  77, 169, 8, 101, 78, 133, 78, 169, 0, 101, 79, 133
5120 DATA  79, 76, 16, 11, 96, 72, 138, 72, 152, 72, 169, 2
5130 DATA  141, 40, 10, 162, 18, 104, 32, 128, 11, 232, 104, 32
5140 DATA  128, 11, 162, 31, 104, 76, 128, 11, 142, 0, 214, 44
5150 DATA  0, 214, 16, 251, 141, 1, 214, 96, 72, 138, 72, 162
5160 DATA  18, 104, 32, 128, 11, 232, 104, 32, 128, 11, 162, 31
5170 DATA  142, 0, 214, 44, 0, 214, 16, 251, 173, 1, 214, 133
5180 DATA  254, 96
```

(PRG 1.28.A) POKE, PEEK und ERASE

```
10 FOR X= 6144 TO 6182
20 READ A:CS=CS+A:POKE X,A
30 NEXT X
40 IF CS<> 3411 THEN PRINT "DATA ERROR IN 40"
50 DATA  72, 138, 72, 152, 72, 169, 2, 141, 40, 10, 162, 18
60 DATA  104, 32, 27, 24, 232, 104, 32, 27, 24, 162, 31, 104
70 DATA  76, 27, 24, 142, 0, 214, 44, 0, 214, 16, 251, 141
80 DATA  1, 214, 96
85 CS=0
90 FOR X= 6656 TO 6697
100 READ A:CS=CS+A:POKE X,A
110 NEXT X
120 IF CS<> 4356 THEN PRINT "DATA ERROR IN 120"
130 DATA  72, 138, 72, 162, 18, 104, 32, 30, 26, 232, 104, 32
140 DATA  30, 26, 162, 31, 142, 0, 214, 44, 0, 214, 16, 251
150 DATA  173, 1, 214, 133, 254, 96, 142, 0, 214, 44, 0, 214
160 DATA  16, 251, 141, 1, 214, 96
165 CS=0
170 FOR X= 6400 TO 6453
180 READ A:CS=CS+A:POKE X,A
190 NEXT X
200 IF CS<> 6426 THEN PRINT "DATA ERROR IN 200"
210 DATA  162, 64, 169, 0, 160, 0, 133, 254, 72, 138, 72, 162
220 DATA  18, 165, 254, 32, 42, 25, 232, 152, 32, 42, 25, 162
230 DATA  31, 169, 0, 32, 42, 25, 104, 170, 104, 200, 208, 228
240 DATA  230, 254, 202, 208, 223, 96, 142, 0, 214, 44, 0, 214
250 DATA  16, 251, 141, 1, 214, 96
```

(PRG 1.28.B) Sinuskurven-Plotter

```
10 REM SINUSKURVEN PLOT PROGRAMM
20 A=DEC("D600"):B=A+1
30 REM PLOT:  SYS DEC("1800"),BYTE,LO,HI
40 REM ERASE: SYS DEC("1900")
50 REM PEEK:  SYS DEC("1A00"),LO,HI:PRINT PEEK(254);
60 :
70 POKE A,25:POKE B,128:POKE A,20:POKE B,0
80 SYS DEC("1900")
110 FOR X=0 TO 639
120 Y=INT(SIN(X/10)*100)+100
130 GOSUB 170
140 NEXT X
150 RETURN
170 REM PLOT
180 AN=80*Y
190 Z1=INT(X/8)
200 AD=AN+Z1: HI=INT(AD/256): LO=AD-256*HI
210 SYS DEC("1A00"),LO,HI
220 PE=PEEK(254)OR2↑(7-(X-Z1*8))
230 SYS DEC("1800"),PE,LO,HI
240 RETURN
```

(PRG 1.29.A) Kopfstehender Zeichensatz

```
10 FOR X=0 TO 511
20 FOR X2=0 TO 3
30 A1=8199+8*X-X2: H1=INT(A1/256): L1=A1-256*H1
40 SYS DEC("1A00"),L1,H1
50 W1=PEEK(DEC("FE"))
60 A2=8192+8*X+X2: H2=INT(A2/256): L2=A2-256*H2
70 SYS DEC("1A00"),L2,H2
80 W2=PEEK(DEC("FE"))
90 SYS DEC("1800"),W2,L1,H1
100 SYS DEC("1800"),W1,L2,H2
110 NEXT X2
120 NEXT X
```

(PRG 1.29.B) Kopfstehender Zeichensatz für 16*8-Matrix

```
10 FOR X=0 TO 511
20 FOR X2=0 TO 7
30 A1=8207+16*X-X2: H1=INT(A1/256): L1=A1-256*H1
40 SYS DEC("1A00"),L1,H1
50 W1=PEEK(DEC("FE"))
60 A2=8192+16*X+X2: H2=INT(A2/256): L2=A2-256*H2
70 SYS DEC("1A00"),L2,H2
80 W2=PEEK(DEC("FE"))
90 SYS DEC("1800"),W2,L1,H1
100 SYS DEC("1800"),W1,L2,H2
110 NEXT X2
120 NEXT X
```

(PRG 2.2.A) Tortengrafik

```
10 REM TORTENGRAPHIK
20 GRAPHIC 0 ,1
30 INPUT"WIEVIEL SEGMENTE";N
40 DIM A(N)
45 DIM P(N)
50 DIM T$(N)
60 FOR I=1 TO N
70 PRINT I". SEGMENT"
80 INPUT "WERT ";A(I)
90 P=P+A(I)
100 INPUT "TEXT ";T$(I)
110 NEXTI
120 INPUT "NOCH AENDERUNGEN (J/N) ";A$
130 IF A$="J" THEN BEGIN
140 FORI=1TON
150 PRINT I".  "T$(I),A(I)
160 NEXT I
170 INPUT "SEGMENT NUMMER ";I
180 P=P-A(I)
190 INPUT "WERT ";A(I)
200 P=P+A(I)
210 INPUT "TEXT ";T$(I)
220 GOTO 120
230 BEND
240 REM TORTE ZEICHNEN
250 GRAPHIC1,1
260 COLOR1,1
270 CIRCLE ,160,100,80
280 G=0
290 FORI= 1TON
300 P(I)=A(I)/P*100
310 G=G+P(I)
320 CIRCLE ,160,100,80,0,0,80,G*3.6
330 IF I/2<>INT (I/2)THEN520
340 REM SEGMENT AUSFUELLEN
350 COLOR1,I/2+.5
360 CIRCLE ,160,100,80,0,0,40,G*3.6-P(I)*1.8
370 X=RDOT(0)
380 Y=RDOT(1)
390 PAINT,X+2,Y+1
400 NEXTI
410 REM TEXTE AUSDRUCKEN
420 G=0
430 FORI=1TON
440 G=G+P(I)
450 COLOR1,I/2+.5
460 CIRCLE,160,100,80,0,89,90,G*3.6-P(I)*1.8
470 COLOR1,1
480 X=RDOT(0)/8
490 Y=RDOT(1)/8
500 IF X<16 THEN X=X-LEN (T$(I))
510 CHAR ,X,Y,T$(I)+STR$(INT(10*P(I)+.5)/10)
520 NEXTI
530 GETKEY A$
540 GRAPHIC0
550 PRINT"NEUE GRAPHIK"
555 PRINT "ALTE GRAPHIK"
560 GETKEYA$
570 IF A$="N" THEN RUN
580 IF A$="A" THEN 120
```

(PRG 2.3.A) Balkengrafik

```
10 REM BALKENGRAFIK
20 GRAPHIC 0
30 SCNCLR
40 DIM A(8)
50 DIM F(8)
60 INPUT "UEBERSCHRIFT";U$
70 U$="      "+U$
80 FOR I=LEN(U$)TO 39
90 U$=U$ +" "
100 NEXT I
110 INPUT "WIEVIEL DATEN (1-8)";D
120 IF D<1 OR D>8 THEN 110
130 FOR I=1 TO D
140 PRINT I". BALKEN"
150 INPUT"GROESSE       ";A(I)
160 IF A(I)>MAX THEN MAX= A(I)
170 INPUT "FARBE (1-16)";F(I)
180 IF F(I)<1 OR F(I)>16 THEN 170
190 NEXT I
200 PRINT "STIMMEN DIE DATEN ?"
210 GETKEY A$
220 IF A$="N" THEN RUN
230 GRAPHIC 1
240 SCNCLR
250 COLOR 1,1
260 CHAR ,0,0,U$,1
270 FORI=1TOD
280 A(I)=190-A(I)/MAX*170
290 COLOR 1,F(I)
300 BOX ,I*38-24,A(I),I*38,190,0,1
310 DRAW,I*38-24,A(I)TOI*38-18,A(I)-6TOI*38+6,A(I)-6TOI*38+6,184 TO I*38,190
320 DRAW ,I*38,A(I)TOI*38+6,A(I)-6
330 NEXT I
340 GETKEY A$
350 GRAPHIC 0
360 SCNCLR
370 PRINT "(A) ALTE GRAFIK"
380 PRINT "(N) NEUE GRAFIK"
390 PRINT "(X) EXIT"
400 GETKEY A$
410 IF A$="X"THEN END
420 IF A$="A"THEN GRAPHIC1:GOTO 340
430 IF A$="N"THEN RUN
440 GOTO 370
```

(PRG 2.4.A) Funktionsplotter

```
10 REM FUNKTIONSPLOTTER
20 GRAPHIC 0,1
30 DEF FNY(X)=SIN(X)
40 INPUT "AUSSCHNITT ANFANG ";A
50 INPUT "AUSSCHNITT ENDE   ";E
60 IF A=>E THEN 40
70 INPUT "KLEINSTER Y-WERT  ";A1
80 INPUT "GROESSTER Y-WERT  ";E1
90 IF A1=>E1THEN 70
100 S=(E-A)/320
110 S2=(E1-A1)
120 GRAPHIC1,1
130 FORI=0TO 319
140 A=A+S
150 X=200-(FNY(A)-A1)/S2 *199
160 IFX=<199ANDX=>0THENDRAW1,I,X
170 NEXT I
180 GETKEY A$
190 GRAPHIC 0
```

(PRG 2.5.8.A) Sichern des Window-Inhalts

```
10 REM WINDOW INHALT SICHERN
20 GRAPHIC 0,1
30 REM BILDSCHIRM VOLLSCHREIBEN
40 FORI=1024TO2024
50 POKE I,J
60 J=J+1
70 IFJ>255THENJ=0
80 NEXT I
90 X1=  5
100 X2=12
110 X3=35
120 X4=22
130 GOSUB 60000
140 REM WINDOW EINGESCHALTEN
150 WINDOW X1,X2,X3,X4,1
160 INPUT "TEXT";A$
170 REM NORMALBILDSCHIRM
180 WINDOW  0, 0,39,24,0
190 GOSUB 60090
200 GETKEY B$
210 GRAPHIC 0,1
220 END
60000 REM INHALT SPEICHERN
60010 DIM X(350)
60020 FOR I=X2 TO X4
60030 FOR J=X1 TO X3
60040 X(Z) = PEEK (I*40 + J +1024)
60050 Z=Z+1
60060 NEXT J
60070 NEXT I
60080 RETURN
60090 REM INHALT ZURUECKSETZEN
60100 Z=0
60110 FOR I=X2 TO X4
60120 FOR J=X1 TO X3
60130 POKE I*40 + J +1024 ,X(Z)
60140 Z=Z+1
60150 NEXT J
60160 NEXT I
60170 RETURN
```

(PRG 2.5.8.B) BASIC-Loader zur Sicherung des Window-Inhalts

```
0 GOTO 150
10 REM TESTPROGRAMM
20 GRAPHIC 0
30 FOR I=1024 TO 2023
40 POKE I,A
50 A=A+1
60 IF A>255 THEN A=0
70 NEXT I
80 WINDOW 10,10,20,20
90 SYS 5120
100 PRINT "□"
110 GETKEYA$
120 PRINT A$;
130 IF A$<>CHR$(13)THEN 110
140 SYS 5120
150 WINDOW 0,0,39,24
160 GRAPHIC 5
170 END
180 REM ROUTINE SAVE WINDOW
190 FOR I= 5120 TO 5227
200 READA
210 S=S+A
220 POKE I,A
230 NEXT I
240 IF S<>16107 THEN BEGIN
250 PRINT"?FEHLER IN DATAS"
260 END
270 BEND
280 GOTO 10
290 DATA165,231, 24,229,230,133,255,230
300 DATA255,169,  4,133,252,162,  0,138
310 DATA228,229,240, 11,232, 24,105, 40
320 DATA144,246,230,252, 76, 16, 20, 24
330 DATA101,230,144,  2,230,252,133,251
340 DATA133,253,165,252,105, 17,133,254
350 DATA160,255,200,173,107, 20,208, 26
360 DATA177,251,145,253,196,255,208,242
370 DATA228,228,232,144, 20,173,107, 20
380 DATA208,  4,238,107, 20, 96,206,107
390 DATA 20, 96,177,253,145,251, 76, 60
400 DATA 20,165,251, 24,105, 40,144,  4
410 DATA230,252,230,254,133,251,133,253
420 DATA 76,48, 20,0
```

(PRG 2.5.9.A) Simulation mehrerer Windows

```
10 REM WINDOW INHALT SICHERN
15 DIM X(40,30)
20 GRAPHIC 0,1
30 REM BILDSCHIRM VOLLSCHREIBEN
40 FORI=1024TO2024
50 POKE I,J
60 J=J+1
70 IFJ>255THENJ=0
80 NEXT I
85 REM WINDOWS ERZEUGEN
90 FORK=0TO39
100 X1=INT(RND(1)*25)+5
110 X2=INT(RND(1)*14)+3
120 GOSUB 60000
130 WINDOW XI,X2,X1+4,X2+3
140 FORW=1TO 20
150 PRINT CHR$(K+48);
160 NEXT W
165 NEXT K
170 REM BILDSCHIRM WIEDER ERSTELLEN
180 FOR K=38TO 0STEP-1
190 X1=X(K,29)
200 X2=X(K,30)
210 WINDOWX1,X2,X1+4,X2+3
220 GOSUB 60090
230 NEXT K
240 GETKEY A$
250 GAPHIC0,1
260 END
60000 REM INHALT SPEICHERN
60010 Z=0
60020 FOR I=X2 TO X2+3
60030 FOR J=X1 TO X1+4
60040 X(K,Z) = PEEK (1*40 + J +1024)
60050 Z=Z+1
60060 NEXT J
60070 NEXT I
60074 X(K,29)=X1
60076 X(K,30)=X2
60080 RETURN
60090 REM INHALT ZURUECKSETZEN
60100 Z=0
60110 FOR I=X2 TO X2+3
60120 FOR J=X1 TO X1+4
60130 POKE I*40 + J +1024 ,X(K,Z)
60140 Z=Z+1
60150 NEXT J
60160 NEXT I
60170 RETURN
```

(PRG 2.6.A) Zeichenkopierer

```
10 REM ZEICHEN KOPIEREN
20 INPUT "WELCHES SPRITE";S
30 IF S<1 ORS>8 THEN 20
40 INPUT "ZEILE (0-13) ";Z
50 IF Z<0 ORZ>13THEN 40
60 INPUT "SPALTE (0-2) ";Y
70 IF Y<0 ORY>2THEN 60
80 REM SPRITE LOESCHEN
90 FORI=0TO62
100 B$=B$+CHR$(0)
110 NEXT I
120 SPRSAV B$,S
130 IF Z=0AND Y=0 THEN 170
140 FORI=1TOZ*3+Y
150 A$=A$+CHR$(0)
160 NEXT I
170 INPUT "CODE DES ZEICHENS ";C
180 FORI=C*8TOC*8+7
190 A=0
200 FORJ=0TO 7
210 BANK 14
220 F=PEEK(53248+I)
230 IF (F AND 2↑J)=2↑J THEN A=A+2↑J
240 NEXT J
250 A$=A$+CHR$(A)+CHR$(0)+CHR$(0)
260 NEXT I
270 SPRSAV A$,S
280 REM SPRITE IM GENERATOR
290 POKE 842,48+S
300 POKE 843,13
310 POKE 208,2
320 SPRDEF
```

(PRG 2.6.B) Zeichen kopieren und vergrössern

```
10 REM ZEICHEN KOPIEREN UND VERGROESSERN
20 INPUT "WELCHES SPRITE";T
30 IF T<1OR T>8THEN 20
40 INPUT "ZEILE (0-5) ";Z
50 IF Z<0ORZ>5 THEN 40
60 INPUT "SPALTE (0-1) ";Y
70 IF Y<0ORY>1 THEN 60
80 REM SPRITE LOESCHEN
90 FORI=0TO62
100 B$=B$+CHR$(0)
110 NEXT I
120 SPRSAV B$,T
130 IF Z=0 AND Y=0 THEN 170
140 FORI=1TOZ*3+Y
150 A$=A$+CHR$(0)
160 NEXT I
170 INPUT "CODE DES ZEICHENS ";S
175 REM ZEICHEN KOPIEREN
180 FORI=S*8TOS*8 +7
190 A(0)=0
200 A(1)=0
210 FORJ=0 TO3
220 FORQ=0 TO1
230 BANK 14
240 F=PEEK(53248+I)
250 IF(FAND2↑(J+Q*3))=2↑(J+Q*3)THEN A(Q)=A(Q)+2↑(J*2)+2↑(J*2+1)
260 NEXT Q
270 NEXT J
280 A$=A$+CHR$(A(1))+CHR$(A(0))+CHR$(0)
290 A$=A$+CHR$(A(1))+CHR$(A(0))+CHR$(0)
300 NEXT I
310 SPRSAV A$,T
320 REM SPRITE IM GENERATOR AKTIVIEREN
330 POKE 842,48+T
340 POKE 843,13
350 POKE208,2
360 SPRDEF
```

(PRG 2.6.1.A) Design im Listing

```
10 REM DESIGN IM LISTING 1
20 INPUT "WIEVIEL SPRITES";S
30 POKE 53296,1
40 FORT=1TOS
50 A$=""
60 FORG=0TO20
70 READ B$
80 IFLEN(B$)<24THEN B$=B$+".":GOTO110
90 FORI=0TO2
100 A=0
110 FORJ=0TO7
120 IF MID$(B$,I*8+J+1,1)="*"THENA=A+2↑(7-J)
130 NEXTJ
140 A$=A$+CHR$(A)
150 NEXT I
160 NEXTG
170 SPRSAV A$,T
180 SPRITET,1,T,1,1,1,0
190 MOVSPR T,T*10#T
200 NEXT T
210 POKE 53296,0
1000 REM 012345678901234567890123
1010 DATA*********
1020 DATA*********
1030 DATA.******
1040 DATA..******
1050 DATA...******
1060 DATA....******
1070 DATA....************
1080 DATA....**************
1090 DATA.....****************
1100 DATA......****************
1110 DATA.......******************
1120 DATA......****************
1130 DATA.....****************
1140 DATA....**************
1150 DATA....************
1160 DATA....******
1170 DATA...******
1180 DATA..******
1190 DATA.******
1200 DATA********
1210 DATA********
1220 REM 012345678901234567890123
```

(PRG 2.6.1.B) Design im Listing 2

```
10 REM DESIGN IM LISTING 2
20 INPUT "WIEVIEL SPRITES";S
30 FORT=1TOS
40 FORG=0TO62STEP 3
50 B$=""
60 FORI=0TO 2
70 FORJ=7TO0STEP-1
80 IF(PEEK(3520+G+I+T*64)AND2↑J)=2↑JTHEN B$=B$+"*":ELSEB$=B$+"."
90 NEXTJ
100 NEXT I
110 PRINT STR$(10000+G+100*T);" DATA";B$
120 PRINT "GOTO 170□□□";:REM 3*CURSOR HOCH
130 POKE 842,13
140 POKE 843,13
150 POKE 208,2
160 END
170 NEXT G
180 NEXT T
190 REM DATEN AB 10100
```

(PRG 2.6.2.A) Sprite-Handling

```
10 REM *************** SPRITE-HANDLING
20 DIM B(600)
30 DIM A(62)
40 REM *************** MENUE
50 INPUT "FARBE 1";F
60 INPUT "FARBE 2";D
70 INPUT "FARBE 2";E
80 SPRCOLORD,E
90 PRINT "1 : SPIEGELN"
100 PRINT "2 : SPIEGELN AN L■NGSACHSE"
110 PRINT "3 : DREHEN UM 180 GRAD"
120 PRINT "4 : SPRITE ANSEHEN"
130 PRINT "5 : INVERTIEREN"
140 PRINT "6 : UM 90 GRAD DREHEN"
150 PRINT "7 : UM 270 GRAD DREHEN"
160 INPUT "BEFEHL ";B
170 IF B<1 OR B>7 THEN GOTO 10
180 INPUT "SPRITE-NUMMER ";S
190 IF S<1OR S>9THEN GOTO 180
200 IF B<5THEN INPUT"MULTICOLOR (JA=1/NEIN=0) ";M
210 :ELSE M=0
220 POKE53296,1
230 IF B>3THEN 270
240 FORI=0TO 62
250 A(I)=PEEK(3520+S*64+I)
260 NEXT I
270 ON B GOSUB 360,430,560,420,600,830,650
280 REM *************** SPRITE ZEIGEN
290 POKE 53296,0
300 GRAPHIC0,1
310 SPRITE S,1,F,1,1,1,M
320 MOVSPR S,100,100
330 GETKEY A$
340 SPRITE S,0
350 GOTO 90
360 REM *************** SPIEGELN
370 FORI=0TO 60 STEP 3
380 FORJ=0TO 2
390 POKE 3520+S*64+I+J,A(60-I+J)
400 NEXT J
410 NEXT I
420 RETURN
```

```
430 REM *************** SPIEGELN AN LAENGSACHSE
440 FORI=0TO 60STEP3
450 FORJ=0TO2
460 W=0
470 FORX=MTO7STEPM+1
480 IF(A(I+2-J)AND2↑X)=2↑XTHEN W=W+2↑(7-X)
490 IF M=1AND(A(I+2-J)AND2↑(X-1))=2↑(X-1)THEN W=W+2↑(7-(X-1))
500 NEXT X
510 A(I+2-J)=W
520 POKE 3520+S*64+I+J,A(I+2-J)
530 NEXTJ
540 NEXTI
550 RETURN
560 REM *************** UM 180 GRAD DREHEN
570 GOSUB 360
580 B=2
590 GOTO 240
600 REM ************** INVERTIEREN
610 FORI=0TO62
620 POKE 3520+S*64+I,255AND(-PEEK(3520+S*64+I)-1)
630 NEXT I
640 RETURN
650 REM *************** UM 270 GRAD DREHEN
660 FORI= 0TO 20
670 FORJ=0TO 2
680 FORG=0TO7
690 B((I*3+J)*8+7-G)=(PEEK (3520+S*64+I*3+J) AND 2↑G)/2↑G
700 NEXT G
710 POKE 3520+S*64+I*3+J,0
720 NEXT J
730 NEXT I
740 FORI=0TO 2
750 FORJ=0TO 20
760 FORG=0TO7
770 IF B(I*192+J+(7-G)*24)=0THEN790
780 POKE 3520+S*64+I+(20-J)*3,PEEK(3520+S*64+I+(20-J)*3)OR 2↑G
790 NEXT G
800 NEXT J
810 NEXT I
820 RETURN
830 REM ******************** UM 90 GRAD DREHEN
840 GOSUB 650
850 B=2
860 GOTO 240
```

(PRG 3.2.A) Listing-Konverter

```
5 PRINT"□BITTE WARTEN..."
10 DIM TK$(255), FE$(29), CE$(8), SZ$(159)
20 FOR Z=0 TO 125: READ WE$, A$: TK$(DEC(WE$))=A$: NEXT Z
22 FOR Z=0 TO 27: READ WE$, A$: FE$(DEC(WE$))=A$: NEXT Z
24 FOR Z=0 TO 7: READ WE$, A$: CE$(DEC(WE$))=A$: NEXT Z
26 FOR Z=0 TO 25: READ WE$, A$: SZ$(DEC(WE$))=A$: NEXT Z
30 AN=7168
40 PRINT"□'B'ILDSCHIRM ODER 'D'RUCKER?"
42 DR=0: GETKEY A$: IF A$="D" THEN DR=1: GOTO 50
44 IF A$<>"B" THEN 40
50 PRINT"■'G'ROSS- ODER 'K'LEINSCHRIFT?"
52 GETKEY A$: IF A$="K" THEN PRINT CHR$(14): GOTO 60
54 IF A$<>"G" THEN PRINT"□";: GOTO 50
60 INPUT"
NAME DES PROGRAMMES";NA$
70 IF DR THEN CLOSE4: OPEN 4,4,0
72 PRINT: PRINT NA$: PRINT
73 IF DR THEN PRINT#4: PRINT#4,NA$: PRINT#4
100 FL=0: AN=AN+3
102 LO=PEEK(AN): AN=AN+1: HI=PEEK(AN): ZE=LO+256*HI
105 PRINT ZE;" ";
106 IF DR THEN PRINT#4,ZE;" ";
110 AN=AN+1
115 WE=PEEK(AN): IF WE=0 THEN 1000
120 IF WE=32 THEN 2000
122 IF WE=34 THEN 7000
123 IF WE=58 THEN 8000
125 IF WE=DEC("FE") THEN 3000
130 IF WE=DEC("CE") THEN 4000
135 IF FF THEN 6000
140 GOTO 5000
999 :
1000 REM NEUE ZEILE
1009 IF DR THEN PRINT#4
1010 PRINT: IF PEEK(AN+1)=0 AND PEEK(AN+2)=0 THEN CLOSE 4: END
1020 GOTO 100
1999 :
2000 REM LEERZEICHEN
2010 IF FF THEN 6030
2020 IF FL THEN PRINT" ";
2021 IF DR AND FL THEN PRINT#4," ";
2030 GOTO 110
2999 :
3000 REM $FE
3010 AN=AN+1: WE=PEEK(AN)
3020 PRINT FE$(WE);" ";
3021 IF DR THEN PRINT#4,FE$(WE);" ";
3030 GOTO 110
3999 :
```

```
4000 REM $CE
4010 AN=AN+1: WE=PEEK(AN)
4020 PRINT CE$(WE);" ";
4021 IF DR THEN PRINT#4,CE$(WE);" ";
4030 GOTO 110
4999 :
5000 REM AUSSERHALB VON STRINGS
5005 IF WE=143 THEN FL=1
5010 IF TK$(WE)<>"" THEN 5040
5020 PRINT CHR$(WE);
5021 IF DR THEN PRINT#4,CHR$(WE);
5030 GOTO 110
5040 PRINT " ";TK$(WE);" ";
5041 IF DR THEN PRINT#4," ";TK$(WE);" ";
5050 GOTO 110
5999 :
6000 REM INNERHALB VON STRINGS
6005 IF WE<>255 THEN 6010
6006 PRINT"π";
6007 IF DR THEN PRINT#4,"π";
6008 GOTO 110
6010 IF SZ$(WE)<>"" THEN 6030
6020 PRINT CHR$(WE);
6021 IF DR THEN PRINT#4,CHR$(WE);
6025 GOTO 110
6030 ZA=1
6040 IF PEEK(AN+1)<>WE THEN 6060
6050 AN=AN+1: ZA=ZA+1: GOTO 6040
6060 PRINT CHR$(18);"[";SZ$(WE);STR$(ZA);"]";CHR$(146);
6061 IF DR THEN PRINT#4," [";SZ$(WE);STR$(ZA);"] ";
6070 GOTO 110
6999 :
7000 REM HOCHKOMMA
7010 FF = XOR (FF,255)
7020 PRINT CHR$(34);: POKE 244,0
7021 IF DR THEN PRINT#4,CHR$(34);
7030 GOTO 110
7999 :
8000 REM DOPPELPUNKT
8010 PRINT CHR$(58);" ";
8011 IF DR THEN PRINT#4,CHR$(58);" ";
8020 GOTO 110
9999 :
```

```
10000 DATA "80","END","81","FOR","82","NEXT","83","DATA","84","INPUT#"
10010 DATA "85","INPUT","86","DIM","87","READ","88","LET","89","GOTO"
10020 DATA "8A","RUN","8B","IF","8C","RESTORE","8D","GOSUB","8E","RETURN"
10030 DATA "8F","REM","90","STOP","91","ON","92","WAIT","93","LOAD","94","SAVE"
10040 DATA "95","VERIFY","96","DEF","97","POKE","98","PRINT#","99","PRINT"
10050 DATA "9A","CONT","9B","LIST","9C","CLR","9D","CMD","9E","SYS","9F","OPEN"
10060 DATA "A0","CLOSE","A1","GET","A2","NEW","A3","TAB(","A4","TO","A5","FN"
10070 DATA "A6","SPC(","A7","THEN","A8","NOT","A9","STEP","AA","+","AB","-"
10080 DATA "AC","*","AD","/","AE","↑","AF","AND","B0","OR","B1",">","B2","="
10090 DATA "B3","<","B4","SGN","B5","INT","B6","ABS","B7","USR","B8","FRE"
10100 DATA "B9","POS","BA","SQR","BB","RND","BC","LOG","BD","EXP","BE","COS"
10110 DATA "BF","SIN","C0","TAN","C1","ATN","C2","PEEK","C3","LEN","C4","STR$"
10120 DATA "C5","VAL","C6","ASC","C7","CHR$","C8","LEFT$","C9","RIGHT$"
10130 DATA "CA","MID$","CB","GO","CC","RGR","CD","RCLR","CF","JOY","D0","RDOT"
10140 DATA "D1","DEC","D2","HEX$","D3","ERR$","D4","INSTR","D5","ELSE"
10150 DATA "D6","RESUME","D7","TRAP","D8","TRON","D9","TROFF","DA","SOUND"
10160 DATA "DB","VOL","DC","AUTO","DD","PUDEF","DE","GRAPHIC","DF","PAINT"
10170 DATA "E0","CHAR","E1","BOX","E2","CIRCLE","E3","GSHAPE","E4","SSHAPE"
10180 DATA "E5","DRAW","E6","LOCATE","E7","COLOR","E8","SCNCLR","E9","SCALE"
10190 DATA "EA","HELP","EB","DO","EC","LOOP","ED","EXIT","EE","DIRECTORY"
10200 DATA "EF","DSAVE","F0","DLOAD","F1","HEADER","F2","SCRATCH"
10210 DATA "F3","COLLECT","F4","COPY","F5","RENAME","F6","BACKUP","F7","DELETE"
10220 DATA "F8","RENUMBER","F9","KEY","FA","MONITOR","FB","USING","FC","WHILE"
10230 DATA "FD","UNTIL","FF","π"
11000 DATA "02","BANK","03","FILTER","04","PLAY","05","TEMPO","06","MOVSPR"
11010 DATA "07","SPRITE","08","SPRCOLOR","09","RREG","0A","ENVELOPE"
11020 DATA "0B","SLEEP","0C","CATALOG","0D","DOPEN","0E","APPEND","0F","DCLOSE"
11030 DATA "10","BSAVE","11","BLOAD","12","RECORD","13","CONCAT","14","DVERIFY"
11040 DATA "15","DCLEAR","16","SPRSAV","17","COLLISION","18","BEGIN"
11050 DATA "19","BEND","1A","WINDOW","1B","BOOT","1C","WIDTH","1D","SPRDEF"
12000 DATA "00","RWINDOW","02","POT","03","BUMP","04","PEN","05","RSPPOS"
12010 DATA "06","RSPRITE","07","RSPCOLOR","08","XOR"
13000 DATA "03","RUN STOP","05","WHT","0A","LINE FEED","11","CRSR DOWN"
13010 DATA "12","RVS ON","13","HOME","1C","RED","1D","CRSR RIGHT","1E","GRN"
13020 DATA "1F","BLU","20","SPACE","81","ORNG","90","BLK","91","CRSR UP"
13030 DATA "92","RVS OFF","93","CLR HOME","95","BRN","96","L RED"
13040 DATA "97","D GRY","98","M GRY","99","L GRN","9A","L BLU"
13050 DATA "9B","L GRY","9C","PUR","9D","CRSR LEFT","9E","YEL","9F","CYN"
```

(PRG 3.4.A) Musik per IRQ

```
0 REM MUSIK AUS DEM IRQ
10 FORI=5120TO5198
20 READ A
30 POKE I,A
40 S=S+A
50 NEXTI
60 IF S<>8891THENPRINT"?FEHLER IN DATAS":END
70 DATA120,169, 22,141, 20,  3,169, 20
80 DATA141, 21,  3, 88,160,  0
90 DATA140,243, 20,200,140,241, 20, 96
100 DATA206,241, 20,208, 49, 72,141, 0
110 DATA255,141, 18,212,138, 72,174,243, 20
120 DATA189,  0, 21,141, 14,212,189,  0
130 DATA 22,141, 15,212,189,  0, 23,141
140 DATA241, 20,169, 65,141, 18,212,232
150 DATA 56,224, 99,144,  2,162,  0,142
160 DATA243, 20,104,170,104, 76,101,250
170 BANK 15
180 POKE 54296,15
190 POKE 54288,255
200 POKE 54289,0
210 POKE 54291,15
220 POKE 54292,0
230 N=99:REM 99 NOTEN
240 FORI=0TON -1
250 READ A
260 POKE 5376+I,A
270 READ A
280 POKE 5632+I,A
290 READ A
300 POKE 5888+I,A
310 NEXT I
320 POKE 5185,N
330 SYS 5120
```

```
340 DATA  10,13, 5, 10,13, 5, 10,13,10
350 DATA  10,13, 5,162,14, 5,162,14,10
360 DATA 247,10,10, 10,13, 5, 10,13, 5
370 DATA 103,17, 5,137,19,13,237,21, 5
380 DATA 237,21,10,237,21, 5,237,21,10
390 DATA 137,19,10,137,19,10,103,17,10
400 DATA 103,17,20,237,21, 5,237,21,13
410 DATA  59,23,10,237,21, 5,237,21, 5
420 DATA 237,21,13,137,19,30,237,21, 8
430 DATA 137,19, 8,137,19,15,103,17,15
440 DATA 103,17,30,103,17,13, 10,13, 5
450 DATA  10,13, 5, 10,13,10,162,14,10
460 DATA 103,17, 5, 10,13, 5, 10,13,10
470 DATA  10,13,10,103,17, 5,137,19,13
480 DATA 137,19, 5,237,21,13,237,21,10
490 DATA 137,19, 5,137,19, 5,137,19, 5
500 DATA 103,17,10,103,17, 5,103,17,20
510 DATA 237,21, 5,237,21,13, 59,23,10
520 DATA 237,21, 5,237,21, 5,237,21,13
530 DATA 137,19,30,237,21,10,137,19,10
540 DATA 137,19,10,103,17,15,103,17,30
550 DATA 237,21, 5,137,19,10,103,17, 5
560 DATA 103,17,20,237,21, 5, 20,26,10
570 DATA  69,29, 5, 69,29,10, 20,26,10
580 DATA  20,26, 5, 20,26,10,237,21, 5
590 DATA 237,21, 5,137,19, 5,103,17,10
600 DATA 162,14, 5, 10,13,10,237,21, 5
610 DATA 237,21,10,103,17,30,237,21,5
620 DATA 237,21,10, 59,23, 5, 59,23,10
630 DATA 237,21, 5,237,21, 5,237,21,10
640 DATA 137,19,30,237,21,10,137,19,10
650 DATA 137,19,10,137,19,13,103,17,13
660 DATA 103,17,30,247,10, 5, 10,13, 5
```

(PRG 3.5.A) Echtzeituhr

```
20 REM 0
30 DATA31,248,,63,252,
40 DATA 127,254,,255,255,,240,15,
50 DATA 224,7,,224,7,,224,7,,224,7,
60 DATA 224,7,,224,7,,224,7,,224,7,
70 DATA 224,7,,224,7,,224,7,,224,7,
80 DATA 240,15,,255,255,,127,254,
90 DATA 63,252,,
100 REM 1
110 DATA3,252,,7,254,,7,254,
120 DATA7,254,,7,254,,3,254,
130 DATA,126,,,126,,,126,,,126,,,126,
140 DATA,126,,,126,,,126,,,126,,,126,
150 DATA,126,,,126,,,126,,,126,,,60,,
160 REM 2
170 DATA63,254,,127,255,,127,255,
180 DATA 124,31,,120,15,,48,15,,,15,
190 DATA ,31,,,63,,,126,,,252,,1,248,
200 DATA 3,240,,7,224,,15,160,,31,134,
210 DATA 63,7,,126,7,,252,7,,255,255,
220 DATA127,254,,
230 REM 3
240 DATA63,254,,127,255,,127,255,
250 DATA 124,31,,120,15,,48,15,,,31,
260 DATA ,63,,,127,,,255,,1,255,
270 DATA 1,255,,,255,
280 DATA ,127,,,63,,96,31,,240,15,
290 DATA 240,15,,255,255,
300 DATA 255,255,,127,254,,
310 REM 4
320 DATA ,124,,,252,,1,252,,1,252,
330 DATA 3,252,,3,252,,7,252,,7,252,
340 DATA 15,188,,15,188,,31,60,,31,60,
350 DATA 62,60,,124,60,,248,60,
360 DATA 255,255,,255,255,,255,255,
370 DATA 255,255,,,60,,,60,,
380 REM 5
390 DATA 127,252,,255,255,,255,255,
400 DATA 240,15,,224,15,,224,6,,240,,
410 DATA 255,,,255,224,,127,248,
420 DATA 1,254,,,127,,,31,,,31,,,31,
430 DATA ,31,,96,127,,241,254,,255,248,
440 DATA 255,224,,127,128,,
450 REM 6
460 DATA 127,252,,255,255,,255,255,
470 DATA 240,15,,224,15,,224,6,,240,,
480 DATA 255,,,255,224,,255,248,
490 DATA 248,254,,240,127,,224,31,,224,31,,224,31,
500 DATA 224,31,,240,127,,248,254,,255,248,
510 DATA 255,224,,127,128,,
520 REM 7
530 DATA63,254,,127,255,,127,255,
540 DATA 124,31,,120,15,,48,15,,,15,
550 DATA ,31,,,63,,,126,,,252,,1,248,
560 DATA 3,240,,7,224,,15,160,,31,128,
570 DATA 63,,,126,,,252,,,252,,,252,,,
```

```
580 REM 8
590 DATA 127,254,,255,255,,255,255,
600 DATA 240,15,,224,7,,240,15,
610 DATA 255,255,,255,255,,127,254,
620 DATA 127,254,,254,127,,240,15,
630 DATA 224,7,,224,7,,224,7,,224,7,
640 DATA 240,15,,127,254,,31,248,
650 DATA 15,224,,1,128,,
660 REM 9
670 DATA 1,254,,7,255,,31,255,,127,143,
680 DATA 254,15,,240,15,,240,15,
690 DATA 240,15,,240,15,,254,15,
700 DATA 127,143,,31,254,,7,255,,1,255,
710 DATA ,15,,96,7,,240,7,,240,15,
720 DATA 255,255,,255,255,,127,254,,
730 POKE 56334,PEEK(56334)OR128
740 FOR I=3456 TO 4095
750 READ A
760 POKE I,A
770 NEXT I
780 REM UHRZEIT SETZEN
790 INPUT "STUNDEN ";S
800 IF S>12 THEN S=S-12:GOTO 800
810 IF S<0 THEN790
820 POKE 56331,S+INT(S/10)*6
830 INPUT "MINUTEN ";M
840 IF M<0 OR M>59 THEN 830
850 POKE 56330,M+INT(M/10)*6
860 INPUT "SEKUNDEN";S
870 IF S<0 OR S>59 THEN 860
880 POKE 56329,S+INT(S/10)*6
890 S=0
900 FORI=5120TO5180
910 READ A
920 S=S+A
930 POKE I,A
940 NEXTI
950 IF S<>6300 THEN PRINT "?FEHLER IN DATAS":END
960 INPUT"UHR AKTIVIEREN";A$
970 POKE 56328,0
980 SYS5120
990 DATA 120,169, 13,141, 20,  3,169
1000 DATA  20,141, 21,  3, 88, 96,160
1010 DATA   3,173, 11,220, 41, 31, 76
1020 DATA  26, 20,185,  8,220, 41,240
1030 DATA  74, 74, 74, 74
1040 DATA  72,185,  8,220, 41, 15
1050 DATA  72,136,208,237,173,  8
1060 DATA 220, 72,160,  7,104
1070 DATA  24,105, 54,153,247,  7,136
1080 DATA 208,246, 76,101,250
1090 FORI=1TO7
1100 SPRITE I,1,1,1,0,1,0
1110 MOVSPR I,30+I*20+INT((I-1)/2)*10,100
1120 NEXT I
```

(PRG 3.6.A) Analoguhr

```
10 REM UHRZEIT SETZEN
20 INPUT "STUNDEN ";S
30 IF S>12 THEN S=S-12:GOTO 30
40 IF S<0 THEN20
50 POKE 56331,S+INT(S/10)*6
60 INPUT "MINUTEN ";M
70 IF M<0 OR M>59 THEN 60
80 POKE 56330,M+INT(M/10)*6
90 INPUT "SEKUNDEN";S
100 IF S<0 OR S>59 THEN 90
110 POKE 56329,S+INT(S/10)*6
120 INPUT"UHR AKTIVIEREN";A$
130 POKE 56334,PEEK(56334)OR128
140 POKE 56328,0
150 REM UHR ZEICHNEN
160 GRAPHIC 1
170 SCNCLR
180 COLOR 1,2
200 CIRCLE ,160,100,74
210 CHAR ,19,4,"12"
220 CHAR,20,21,"6"
230 CHAR,28,12,"3"
240 CHAR,11,12,"9"
270 REM STUNDEN UND MINUTEN ZEICHNEN
280 COLOR1,12
290 M=INT(PEEK(56330)/16)*10+(PEEK(56330)AND15)
300 COLOR 1,1
310 CIRCLE,160,100,0,60,40,90,M*6
320 H=((16ANDPEEK(56331))/16)*10+(PEEK(56331)AND15)
330 CIRCLE,160,100,0,60,60,90,H*30+M/2
340 REM  SEKUNDE ZEICHNEN
350 WAIT 56328,8
360 COLOR 1,12
370 CIRCLE,160,100,0,110,0,40,S*6
380 S=INT(PEEK(56329)/16)*10+(PEEK(56329)AND15)
390 COLOR 1,2
400 CIRCLE,160,100,0,110,0,40,S*6
410 IF S=0 THEN170
420 GOTO 350
```

(PRG 3.10.A) Signalton

```
10 REM SIGNALTON
20 FORI= 5165 TO 5234
30 READ A
40 S=S+A
50 POKE I,A
60 NEXT I
70 IF S<>8167 THENBEGIN
80 PRINT "?FEHLER IN DATAS"
90 END
100 BEND
110 SYS 5165
120 DATA169,255,141,  6,212,141, 24,212
130 DATA169,  9,141,  5,212,120,169, 71
140 DATA141,20,3,169,20,141,21,3,88, 96
150 DATA 72,165,231,229,236,201,  2,208
160 DATA 17,205,128, 20,240, 15,141,128
170 DATA 20,169,160,141,  1,212,169, 33
180 DATA208,  5,141,128, 20,169,  0,141
190 DATA  1,212,141,  0,212,141,  4,212
200 DATA104, 76,101,250
```

(PRG 4.1.A) Schutzroutine

```
60000 LI=45
60010 LI=PEEK(LI)+256*PEEK(LI+1)
60020 IF LI=0 THEN PRINT "FERTIG!":END
60030 PRINT CHR$(145);"PASS ";PEEK(BA+2)+256*PEEK(BA+3);
60040 PRINT X$
60050 BA=LI
60060 FOR X=4 TO 8
60070 IF PEEK(LI+X)<>ASC(":")THEN X=8:FL=1
60080 NEXT X
60090 IF FL=1 THEN FL=0:GOTO 60010
60100 POKE LI+4,0:X$=X$+"*"
60110 GOTO 60010
```

(PRG 4.2.A) Zeilennnummern-Roulette

```
60000 BA=PEEK(45)+256*PEEK(46)
60010 LI=PEEK(BA)+256*PEEK(BA+1)
60020 IF LI=0 THEN PRINT "FERTIG !":END
60030 PRINT "GEFUNDENE ZEILE: ";PEEK(BA+2)+256*PEEK(BA+3)
60040 PRINT "(A)ENDERN (W)EITER ODER (E)NDE ??!"
60050 GETKEY A$
60060 IF A$="E" THEN PRINT "OK":END
60070 IF A$="W" THEN 60140
60080 INPUT "NEUE ZEILENNUMMER";ZN
60090 IF ZN<0 OR ZN>65535 THEN 60080
60100 HI=INT(ZN/256):LO=ZN-(256*HI)
60110 POKE BA+2,LO
60120 POKE BA+3,HI
60130 PRINT CHR$(145);
60140 BA=LI
60150 PRINT CHR$(145);CHR$(145);
60160 GOTO 60010
```

(PRG 4.3.A) Manipulation des Zeilen-Links

```
60000 INPUT "1. ZEILENNUMMER";Z1
60010 INPUT "2. ZEILENNUMMER";Z2
60020 BA=45
60030 BA=PEEK(BA)+256*PEEK(BA+1)
60040 IF BA=0 THEN PRINT "ZEILE EXISTIERT NICHT !":END
60050 ZN=PEEK(BA+2)+256*PEEK(BA+3)
60060 IF FL=0 THEN IF ZN=Z1 THEN L1=BA:L2=BA+1:FL=1
60070 IF FL=0 THEN 60030
60080 IF ZN=Z2 THEN POKE L1,PEEK(BA):POKE L2,PEEK(BA+1):END
60090 GOTO 60030
```

(PRG 5.2.A) DATA-Generator

```
10 PRINT"[illegible]D A T A - G E N E R A T O R"
20 PRINT:PRINT:PRINT
30 PRINT"DIESES PROGRAMM SCHREIBT IHNEN";
40 IF PEEK(215)<128 THEN PRINT
50 PRINT"IHREN IDIVIDUELLEN BASIC-LOADER!"
60 PRINT:PRINT
70 PRINT"F O R M A T - E I N G A B E : "
80 PRINT:PRINT
90 INPUT "SPEICHERKONFIGURATION (BANK)";BK
100 INPUT "DATENSATZ LESEN  AB (HEX)";A$
110 INPUT "DATENSATZ LESEN BIS (HEX)";B$
120 A=DEC(A$)-1:B=DEC(B$)
130 :
140 INPUT "BASIC-LOADER AB ZEILENNR.";Z:O=Z
150 INPUT "SCHRITTWEITE DER BASIC-ZEILEN";S
160 INPUT "WIEVIEL DATEN PRO ZEILE";DZ:XY=DZ
170 :
180 GOSUB 1000
190 :
200 LD=B-A
210 PRINT"[illegible]"
220 IF LD<DZ THEN DZ=LD:FL=1
230 LD=LD-DZ
240 PRINT Z;"DATA ";
250 IF DZ>1 THEN FOR W=1 TO DZ-1:ELSE GOTO 290
260 BANK BK:P=PEEK(A+W+(XY*ZE)):PRINT P;"[illegible],";
270 CS=CS+P
280 NEXT W
290 ZE=ZE+1
300 BANK BK:P=PEEK(A+(XY*(ZE-1))+DZ):PRINT P:CS=CS+P
310 :
320 Z=Z+S
330 IF FL=0 THEN PRINT "GOTO 210":ELSE PRINT "GOTO 900"
340 GOTO 800
800 REM TASTATURPUFFER FUELLEN
810 PRINT"[illegible]";
820 FOR X=842 TO 851
830 POKE X,13
840 NEXT X
850 POKE DEC("D0"),10
860 END
900 REM EDITOR
910 PRINT"[illegible]";
920 PRINT O+3*S;"IF CS<>";CS;"THEN PRINT CHR$(7);:LIST"
930 PRINT"DELETE -1300"
940 BANK 15:GOTO 800
1000 REM INIT
1010 PRINT"[illegible]"
1020 PRINT Z;"FOR X=";A+1;"TO";B
1030 PRINT Z+S;"READ A:CS=CS+A:POKE X,A"
1040 PRINT Z+2*S;"NEXT X"
1050 Z=Z+4*S
1060 PRINT "GOTO 200"
1070 GOTO 800
```

(PRG 6.5.A) Datasette als begnadete Musikbox

```
5000 FOR X= 6656 TO 6676
5010 READ A:CS=CS+A:POKE X,A
5020 NEXT X
5030 IF CS<> 2799 THEN PRINT CHR$(7);:LIST
5040 DATA  160, 0, 173, 13, 220, 201, 0, 240, 2, 160, 15, 140
5050 DATA  24, 212, 165, 213, 201, 88, 240, 236, 96
```

(PRG 7.6.A) Belegung der Funktionstasten

```
10 REM 8 FUNKTIONSTASTEN
20 FORI=5120 TO  5190
30 READA
40 S=S+A
50 IF A<>PEEK (I)THEN PRINT A,I ,PEEK(I)
60 NEXTI
70 IF S<>7375 THEN BEGIN
80 PRINT "?FEHLER IN DATAS"
90 END
100 BEND
110 SYS 5120
120 DATA120,169, 13,141,20,3,169,20,141
130 DATA 21,  3, 88, 96, 72,152, 72,165
140 DATA211,168, 41,  8,240, 29,152,205
150 DATA 96, 20,240, 23,141, 96, 20, 10
160 DATA 10, 10, 10,168,185,  0, 20,240
170 DATA 13,201, 13,240, 15, 32,210,255
180 DATA200,208,241,140, 96, 20,104,168
190 DATA104, 76,101,250,141, 74,  3,169
200 DATA   1,133,208,76,55,20
210 REM BELEGUNG DER FUNKTIONSTASTE
220 FOR I=0 TO7
230 B$="                    ":REM 15 LEERZEICHEN
240 PRINT "ALT-";I+1
250 PRINT "TEXTEINGABE: BEENDEN MIT LINE FEED"
260 FORJ=0 TO 14
270 GETKEYA$
280 IF A$=CHR$(10)THEN330
290 IF A$=CHR$(13)THEN330
300 MID$(B$,J+1,1)=A$
310 POKE 5248+J+I*16,ASC(A$)
320 NEXTJ
330 POKE 5248+J+I*16,0
340 PRINT B$
350 INPUT "ZUFRIEDEN (J/N)";A$
360 IF A$="N" THEN 230
370 NEXTI
```

(PRG 7.7.A) Tastaturpiep

```
10 REM TASTATURPIEP
20 FOR I=5120 TO5195
30 READ A
40 S=S+A
50 POKE I,A
60 NEXT I
70 IF S<>8932 THENBEGIN
80 PRINT "?FEHLER IN DATAS"
90 END
100 BEND
110 SYS 5120
120 DATA169,255,141,  6,212,141, 24,212
130 DATA169,  9,141,  5,212,120,169, 26
140 DATA141, 20,  3,169, 20,141, 21,  3
150 DATA 88, 96, 72,165,213,201, 88,240
160 DATA 34,201, 76,208, 12,169,103,141
170 DATA  0,212,169, 17,141,  1,212,208
180 DATA 20,201,  1,240,240,169,103,141
190 DATA  1,212,169, 33,141,  0,212,169
200 DATA 17,208,  2,169,  0,141,  4,212
210 DATA104, 76,101,250
```

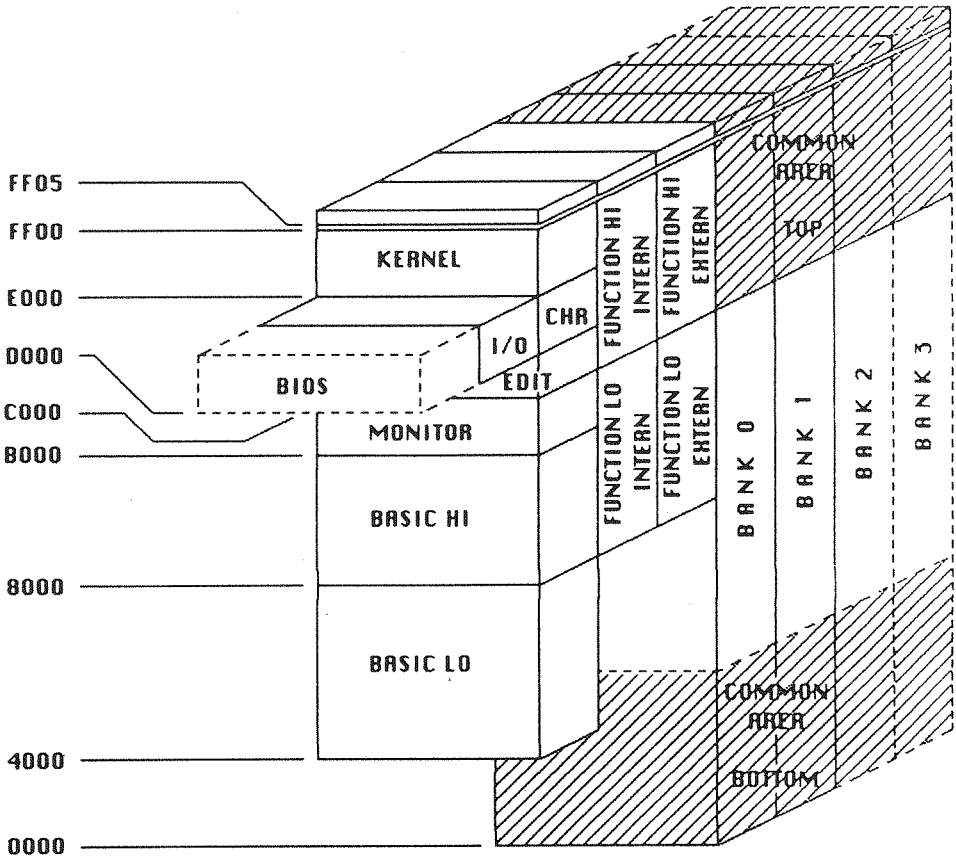
(PRG 7.8.A) Stopfunktion

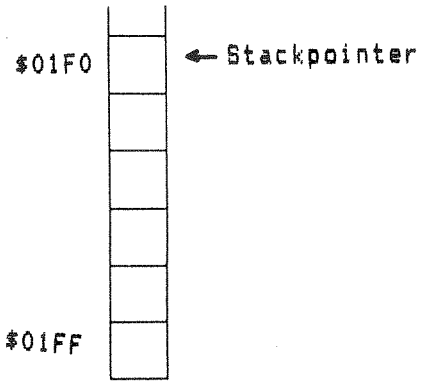
```
10 REM STOPFUNKTION
20 FORI=5120TO 5165
30 READ A
40 S=S+A
50 POKE I,A
60 NEXT
70 IF S<>5361 THEN BEGIN
80 PRINT "?FEHLER IN DATAS"
90 END
100 BEND
110 SYS 5120
140 DATA169, 11,141,  8,  3,169, 20,141
150 DATA  9,  3, 96,165,212,201, 87,208
160 DATA 10,165,212,201, 87,240,250,201
170 DATA 88,240,246,169, 15,133,  2,169
180 DATA 74,133,  3,169;162,133,  4,169
190 DATA  0,133,  5, 76,227,  2
```

(PRG 7.9.A) Belegung der Funktionstasten 9 und 10

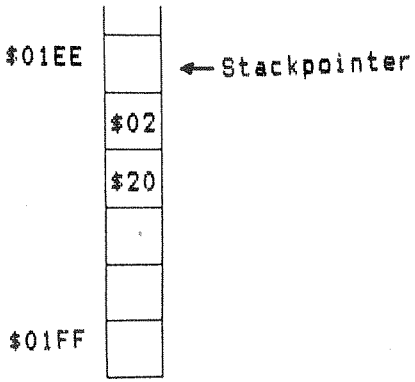
```
10 REM FUNKTIONSTASTEN 9 + 10
20 REM ANFANGSADRESSE ERRECHNEN
30 FORI=4096 TO 4103
40 A=A+PEEK (I)
50 NEXTI
60 PRINT"↑ = RETURN"
70 PRINT CHR$(27);"F"
80 FOR I=0TO1
90 PRINT"FUNKTIONSTASTE " 9+I
100 REM EINGABE
110 GETKEYB$
120 IF B$<>CHR$(13)THEN BEGIN
130 PRINT B$;
140 B=ASC(B$)
150 IF B=94 THEN B=13
160 POKE 4106+A+Z, B
170 Z=Z+1
180 IF A+Z+4106>4351 THEN END
190 GOTO 110
200 BEND
210 PRINT
220 POKE 4104+I,Z
230 A=A+Z
240 Z=0
250 NEXTI
```

(Zeichnung 9.2.a)

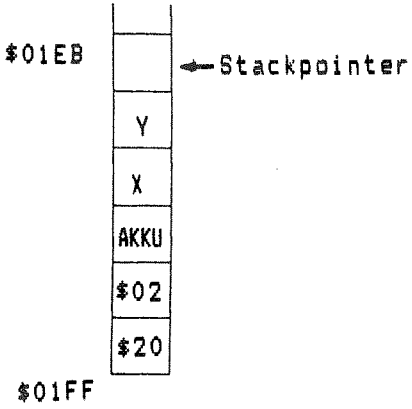




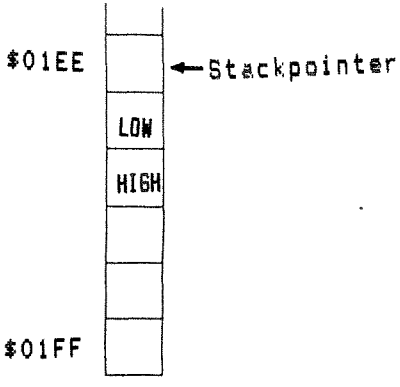
(Zeichnung 11.2.1.a)



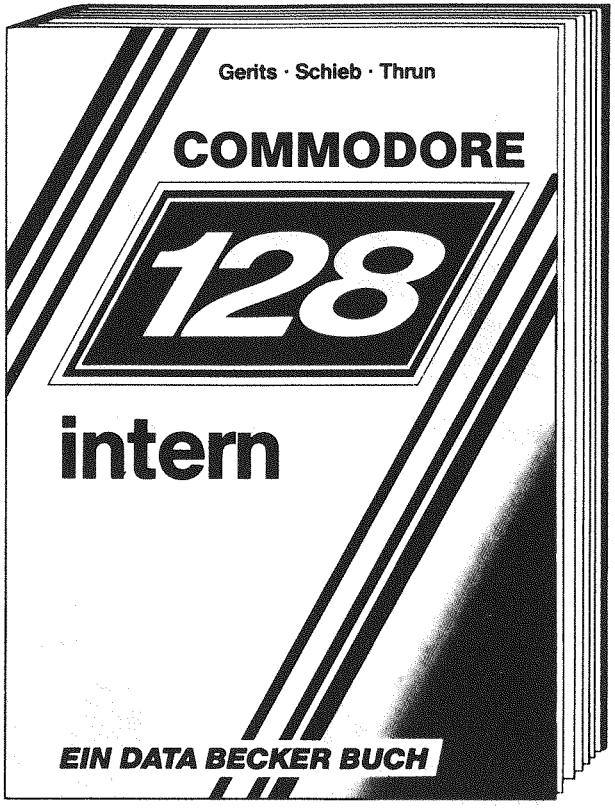
(Zeichnung 11.2.1.b)



(Zeichnung 11.2.1.c)



(Zeichnung 11.2.1.d)



Einführung in das System, Hardware- und Interfacebeschreibung, Erläuterung des VIC-Chips, des VDC (640 x 200 Grafik auf dem 80-Zeichen-Schirm, 28 Zeilen), SID, detaillierte Beschreibung der Memory-Management-Unit (MMU), ein sehr ausführlich dokumentiertes ROM-Listing. Mit sehr vielen hilfreichen Programmbeispielen. Ein Superbuch, wie alle Titel in der INTERN-Reihe!
128 INTERN, über 500 Seiten, DM 69,–

Falls Sie mit dem Commodore 128 in die CP/M-Welt einsteigen wollen, sind Sie hier richtig. Von grundsätzlichen Erklärungen zu Betriebssystem und Speicherung von Zahlen, Schreibschutz oder ASCII, Schnittstellen und Anwendung von CP/M-Hilfsprogrammen. Für Fortgeschrittene: CP/M und Commodore-Format, Erstellen von Submit-Dateien u. v. m. Nutzen Sie die Möglichkeiten von CP/M!

Das CP/M-Buch zum PC128,
ca. 250 Seiten, DM 49,–

Jetzt gibt es das große Floppybuch auch zur 1571 mit einer Einführung für Einsteiger. Arbeiten mit dem PC-128 und BASIC 7.0, sequentiellen und relativen Dateien. Für Fortgeschrittene: Nutzung der Direktzugriffsbefehle, Programme im DOS, wichtige DOS-Routinen und ihre Anwendung und natürlich ein ausführlich dokumentiertes DOS-Listing. Unentbehrlich zum effektiven Einsatz der 1571!

Das große Floppybuch 1571, ca. 300 Seiten, DM 49,–

C COMPILER

C-COMPILER
DM 298,-*

"C" ist die kommende Programmiersprache! Bereits heute arbeiten große Softwarehäuser mit ihr - sogar das bekannte Betriebssystem "UNIX" wurde in "C" geschrieben. Jetzt kann auch der C-64-Anwender diese zukunftsweisende Computersprache nutzen, die bisher nur den "Großen" vorbehalten war. Und zwar in vollem Umfang, denn der C-Compiler von DATA BECKER ermöglicht nicht nur einen Einblick in dieses hochinteressante System, sondern bietet eine Möglichkeit zur professionellen Programmerstellung - eine echte Alternative zu anderen Sprachen. Das C-Paket enthält Editor, Compiler, Dienstprogramme sowie ein ausführlich dokumentiertes Handbuch.

Der C-COMPILER in Stichworten:
EDITOR: Full Screen Editor mit variabler Zeichenbreite, maximal 43 K Textspeicher, 2 Zeichensätzen und komfortablen Textverarbeitungsfunktionen.
COMPILER: Voller Sprachumfang nach Kernigan und Ritchie (außer Bitfeldern), erzeugt direkten Maschinencode, hat 50 KByte Speicher für den objektcode verfügbar, optimiert Ausdrücke und bietet 16(!)-stellige Fließkomma-Arithmetik.
LINKER: Bindet bis zu 7 getrennt compilierte Quellfiles zu einem lauffähigen Maschinenprogramm zusammen, das sowohl vom C-Hauptmenue als auch von BASIC aus gestartet werden kann.
COPY: Diskettendienstprogramm.
Mit dem DATA BECKER "C"-COMPILER steht dem C-64-Anwender jetzt ein mächtiges Softwarewerkzeug zur Verfügung. "C"-Programme lassen sich ohne großen Aufwand auf PC's oder selbst auf Großrechner übertragen. Der DATA BECKER "C"-COMPILER wurde komplett in Deutschland von deutschen Autoren entwickelt. Natürlich mit ausführlichem Handbuch. Programm wird auf Diskette (für Diskettenlaufwerk 1541) geliefert.

* unverbindliche Preisempfehlung. Alle Programme auf Diskette für VC-1541.

ADA-Trainingskurs
DM 198,-*

Diese Programmiersprache der Zukunft, wie seinerzeit COBOL vom Pentagon in Auftrag gegeben, kann jetzt mit dem DATA BECKER-Trainingskurs auch der C-64-Anwender erlernen. Der ADA-Trainingskurs enthält außerdem einen Compiler, der einen umfassenden SUBSET und die wesentlichen Elemente dieser Sprache bietet.

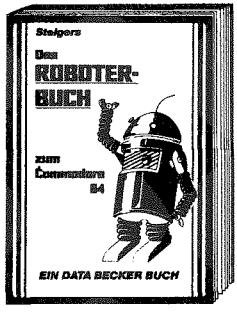
ADA-Trainingskurs in Stichworten:
blockstrukturierte Programme - modularer Aufbau der Programme - ermöglicht die Behandlung von Ausnahmezuständen - lexikalische, syntaktische und semantische Fehlerüberprüfung beim Übersetzen und zur Laufzeit - Nennung der fehlerhaften Zeilennummer führt zu problemloser Fehlersuche - ermöglicht das einfache Einbinden von Maschinenprogrammen - ausgesprochen leichtes Arbeiten mit Programmbibliotheken - Konstanten und Variablendefinitionen von verschiedenen Typen (Integer, Strings etc.) - Assembler erlaubt Kommentare, Benutzung von Labels, verschiedene Zahlenformate, Pseudo-Anweisungen (z.B. ".BYTE", ".MARKE", ".START", ".BLOCK", ".WORD", ".COUNT" etc.) und die Verwendung aller Mnemonics (MOS Standard) - Disassembler ermöglicht die Analyse von Maschinenprogrammen - 62 Schlüsselwörter, trotzdem genug Speicherplatz für eigene Programme vorhanden - Programmdiskette enthält Editor, Übersetzer, Assembler und Disassembler - umfangreiches deutsches Handbuch mit Übungen und Lösungsvorschlägen zu den Themen Textausgabe, Bildschirmsteuerung, Datenobjekte, Datenein- und -ausgabe, Wertzuweisung, Entscheidungen und Schleifen - ausführliche Darstellung und Erläuterung der Grammatikregeln - Grammatikindex.

* unverbindliche Preisempfehlung. Alle Programme auf Diskette für VC-1541.

Diese hochkarätige Einführung in die rechnerunterstützte Konstruktion liefert neben umfassenden Informationen reichlich Konstruktionsbeispiele mit etlichen Programmen. Konkret werden dreidimensionale Zeichnungen und deren Veränderung durch Zoonen, Duplizieren, Spiegeln etc. behandelt, Bausteinprinzip und Macros erklärt sowie darüber hinaus der Aufbau eines eigenen CAD-Systems erarbeitet. Ein brandaktuelles Buch der absoluten Spitzenklasse!

Helft
Einführung in CAD mit dem Commodore 64
302 Seiten, DM 49,-
ISBN 3-89011-067-3

Steigers
Das Roboterbuch zum Commodore 64
ca. 230 Seiten, DM 49,-
erscheint April 1985
ISBN 3-89011-86-X



STAR-TRECK im Wohnzimmer? Dieses packende Buch zeigt, wie man sich einen Roboter ohne großen finanziellen Aufwand selbst bauen kann und welche erstaunlichen Möglichkeiten der C 64 zur Programmierung und Steuerung bietet - anschaulich dargestellt mit vielen Abbildungen und etlichen Beispielen. Dazu ein spannender berblick über die historische Entwicklung des Roboters und eine umfassende Einführung in kybernetische Grundlagen. Unentbehrlich für jeden Roboterfan!

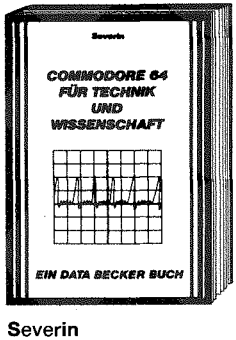
Voß
Einführung in die Künstliche Intelligenz
Mit vielen Programmen für den C 64
395 Seiten, DM 49,-
ISBN 3-89011-081-9

Zentrales Thema aktueller Diskussionen: die Künstliche Intelligenz (KI). Eine ausführliche und interessante Einführung in deren Theorie und Einsatzmöglichkeiten, vom historischen Abriß über die "denkenden" und "lebenden" Maschinen bis zu Anwendungsbeispielen mit Programmen für den Commodore 64. Expertensystem, Such- und Auskunftsprogramm oder selbstlernende Programme werden ebenso dargestellt wie Computer-Kunst oder Simulationen.

Sasse
Compiler - verstehen, anwenden, entwickeln
336 Seiten, DM 49,-
ISBN 3-89011-061-4

Zu den wichtigsten Arbeitsmitteln des Programmierers überhaupt gehören Compiler, deren Grundlagen, Funktionen und Einsatzweise in diesem Buch systematisch erklärt werden. Auch die Entwicklung eines eigenen Compilers, lexikalische, syntaktische und semantische Analyse sowie Codegenerierung sind ausführlich beschrieben. Mit vielen nützlichen Programmen, speziell zugeschnitten auf den Commodore 64 - Pflichtlektüre für jeden ernsthaften Programmierer!

Konkurrenzlos! Dieses Buch enthält nicht nur eine umfangreiche Programmsammlung, sondern ist zuglich qualifiziertes Standardwerk (inklusive Tips und Tricks!) für die anspruchsvolle wissenschaftliche Nutzung des C 64. Mit Sortier- und Mathematikprogramm, Statistik und weiteren interessanten Programmen für Chemie, Physik, Biologie und Elektronik wird der 64er zur wissenschaftlichen Hilfskraft. Ein breites Spektrum, gut und ausführlich dokumentiert.



Severin
Commodore 64 für Technik und Wissenschaft
296 Seiten, DM 49,-
ISBN 3-89011-021-5

Ein unentbehrliches Arbeitsinstrument für den Commodore-64-Anwender! Fachwissen von A-Z bei allen Fragen zur Computerei im allgemeinen und zum 64er im besonderen. Gleichzeitig ein Fachwörterbuch, natürlich mit deutscher Erklärung der englischen Fachbegriffe. Insgesamt eine unglaubliche Vielfalt an Informationen, die grundsätzliches Verständnis ebenso fördern wie fortgeschrittene Programmierung.

Jordan/Schellenberger
Das DATA BECKER Lexikon zum Commodore 64
354 Seiten, DM 49,-
ISBN 3-89011-0013-4